EVOLUTIONARY APPROACHES TO SOLVE THE 3D THERMAL-AWARE FLOORPLANNING PROBLEM USING HETEROGENEOUS PROCESSORS

IGNACIO ARNALDO LUCAS

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería de Computadores
calificado con SOBRESALIENTE en la convocatoria de Junio 2011

Jueves 16 de Junio de 2011

Directores:

J. Ignacio Hidalgo
José L. Ayala
José L. Risco-Martín

Autorización de difusión

Ignacio Arnaldo Lucas

Jueves 16 de Junio de 2011

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "EVOLUTIONARY APPROACHES TO SOLVE THE 3D THERMAL-AWARE FLOORPLANNING PROBLEM USING HETEROGENEOUS PROCESSORS", realizado durante el curso académico 2010-2011 bajo la dirección de J. Ignacio Hidalgo y con la colaboración externa de dirección de José L. Ayala y José L. Risco-Martín en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Director del proyecto:

J. Ignacio Hidalgo:

Resumen en castellano

La integración en 3D es una técnica prometedora para llevar a cabo el proceso de fabricación de futuras arquitecturas multiprocesador. Esta técnica mejora el rendimiento y reduce el cableado obteniendo así un menor consumo global. Sin embargo, la integración en 3D provoca problemas térmicos de gran importancia debidos a la mayor proximidad de elementos que irradian calor, acentuando el impacto de los puntos calientes. Los algoritmos de floorplanning juegan un papel importante en la reducción del impacto térmico, pero no tienen en cuenta el perfil dinámico de las aplicaciones. Este trabajo propone un innovador floorplanner guiado por los pérfiles de consumo de potencia de un conjunto de aplicaciones representativas del ámbito de ejecución. Los resultados muestran que tener en cuenta el perfil dinámico de las aplicaciones en vez del los valores en el caso peor lleva a mejorar la respuesta térmica del chip.

Palabras clave

multiprocesador, floorplanning, restricciones térmicas, perfil dinámico de ejecución, multiobjetivo, algoritmo genético

Abstract

3D integration has become one of the most promising techniques for the integration future multi-core processors, since it improves performance and reduces power consumption by decreasing global wire length. However, 3D integration causes serious thermal problems since the closer proximity of heat generating dies makes existing thermal hotspots more severe. Thermal-aware floorplanners can play an important role to improve the thermal profile, but they have failed in considering the dynamic power profiles of the applications. This work proposes a novel thermal-aware floorplanner guided by the power profiling of a set of benchmarks that are representative of the application scope. The results show how our approach outperforms the thermal metrics as compared with the worst-case scenario usually considered in "traditional" thermal-aware floorplanners.

Keywords

multiprocessor, floorplanning, thermal-aware, profiling, multi-objective, genetic algorithm

Table of Contents

Chapter 1

Introduction

As an introduction to our work, we explain the motivations that led us to study this subject. We present the state of the art of the literature related to the floorplanning problem focusing both on the techniques and heuristics used and in the 2D and 3D representations of the elements that compose a chip. In the last section of the introduction we explain the contributions of our work.

1.1 Motivations

In the last few years semiconductor industry has seen innumerable engineering advances that have permitted a logarithmic growth in the capability of integrated circuits (ICs). This trend was first publicized by Gordon E. Moore in 1965, who suggested that the number of transistors on an IC doubled every two year (his observation is now known as Moore's Law). In fact, huge advances in technology and frequency scaling allowed the majority of computer applications to increase in performance without requiring structural changes or custom hardware acceleration. While these advances continue, their effect on modern applications is not as dramatic as other obstacles such as the memory-wall and the power-wall.

Power density of the microprocessors is increasing with every new process generation since feature size and frequency are scaling faster than the operating voltage [6]. As a result, there has been an increase in maximum chip temperatures because power density directly translates into heat. For example, Pentium 4 chips generate more heat than a kitchen hotplate. If a Pentium 4 chip is allowed to run without a proper cooling system, it catches fire [2]. Intel's projections show that the heat generated by the processors will increase sharply in the coming years, approaching that of the core of a nuclear power plant, unless solutions to this problem can be found [1].

The 3D IC is gaining a lot of interest as a viable solution to help maintain the pace of system demands on scaling, performance, and functionality. The benefits include system-size reduction, performance enhancement due to shorter wire length, power reduction and

the potential for hetero-integration. In the field of Multi-Processor Systems-on-Chip (MPSoCs), only 3D stacks are able to provide the required space for integration. Multiprocessor system-on-chips (MPSoCs) are now widely used in application-specific systems and high-performance computing. They offer performance, design and implementation complexity, reduced power consumption, and thermal benefits over massively superscalar uniprocessor architectures. Nowadays, the primary method of gaining extra performance out of computing systems is to introduce additional resources in a single chip. There are two main approaches that follow this idea, one of them leads to multicore and manycore homogeneous processors which consists in replicating a core several times in a single chip. Multicore and manycore processors have led to an improvement of the overall performance while reducing the working frequency. As a consequence, the power dissipation of the chip has remained in acceptable levels. The other approach introduces specialized resources to achieve specific tasks leading to heterogeneous computing systems. This allows a designer to use multiple types of processing elements, each one able to perform the tasks that it is best suited for. The combination of these two approaches is considered to be a future trend in computer design. As a consequence of this combination, new problems emerge such as binary incompatibility, just-in time compilation, the need of efficient scheduling techniques for heterogeneous systems etc. Other traditional problems like floorplanning persist and are more and more relevant with the increase of computing elements in a single die.

Floorplanning has been proved to be a crucial step in VLSI (Very Large Scale Integration) physical design. In fact, aggressive performance improvements have resulted in a dramatic power consumption increase and a loss of circuit performance and reliability. As explained in [21] MOS current drive capability decreases approximately 4% for every 10°C temperature increase, interconnect delay increases approximately 5% for every 10°C increase while the leakage current increases exponentially with the temperature. Thermal aware floor planning is finding an optimum floor plan by optimizing the cost function consisting of area, wire length, and temperature. The objective of the problem is to minimize the chip area, minimize the wire length, and minimize the maximum temperature of the chip. Thermal aware floor planning can be used as one of the methods for decreasing the maximum temperature of the chip. Cooling of the blocks in a floor plan arises due to lateral spreading of heat through silicon blocks [29]. If a hot block is placed besides cooler blocks, lateral spreading of heat takes place. As a result, the temperature of the hot block is reduced.

A common limitation of the previous methods of 3D floorplanning is that they are focused on area and/or wire length minimization with or without thermal considerations. This can be a serious limitation as modern floorplanners often have to work with a fixed die size constraint, or with a fixed outline constraint in low-level design of hierarchical floorplanning flow [3].

However, all recently developed thermal-aware tools deploy temperature estimation techniques only on a single power profile representing power profiles of all inputs and all applications (e.g. using average or peak power profile). Different applications lead to different

dynamic power profiles of the blocks. Most of the existing work use either average power or peak power per block of the applications for simulating temperature, without analyzing the impact of this assumption.

1.2 Related work

Power consumption has become a critical issue in VLSI physical design. In fact, floorplanning techniques that take thermal data into account are more and more relevant and, as a result, the floorplanning problem has become a common research topic. The impact of the floorplanning on the thermal distribution of real microprocessor-based systems is analyzed in [17], where the placement of components for Alpha and Pentium Pro is evaluated.

Some initial works on thermal aware floorplanning [8] propose a combinatorial optimization problem to model our problem. However, the simplification of the considered floorplan and the lack of a real experimental framework motivated the further research on the area. Thermal placement for standard cell ASICs is a well researched area in the VLSI CAD community, where we can find works as [7].

In the area of floorplanning for microprocessor-based systems, some authors consider the problem at the microarchitectural level [29], where it is shown that significant peak temperature reduction can be achieved by managing lateral heat spreading through floorplanning.

Thermal-aware floorplanning for 3D stacked systems has also been investigated. Cong [11] proposed a thermal-driven floorplanning algorithm for 3D ICs, which is a natural extension of his previous work on 2D. In [19], Healy et al. implemented a multi-objective floorplanning algorithm for 2D and 3D ICs, combining linear programming and simulated annealing. Recent works as [12] also propose combinatorial techniques to tackle the problem of thermal-aware floorplanning in 3D multi-processor architectures.

Other works [21] use genetic algorithms to demonstrate how to decrease the peak temperature while generating floorplans with area comparable to that achieved by traditional techniques. In "A Slicing Structure Representation for the Multi-layer Floorplan Layout Problem", a genetic algorithm is used to solve the multiple layer floorplanning problem. Its main contribution is a three dimensional slicing structure representation. A floorplan is encoded in normalized Polish expressions with horizontal, vertical, and lateral cuts. In order to evaluate a floorplan, the authors propose to break down the 3D slicing structure. They define a slicer algorithm which accepts a 3D floorplan and the maximum number of layers, and returns a slicing structure for each layer. With this representation, a genetic algorithm is used to minimize whether the overall total area or the balanced area. The advantage of this representation is that the crossover and mutation operators are very fast, on the other hand, polish representations are not suitable for thermal-aware floorplanning. Nevertheless, slicing structures remain one of the most used representations for the floorplanning prob-

lem. In "Thermal-Aware Floorplanning Using Genetic Algorithms", Hung et al. present a thermal-aware floorplanning framework based on a genetic algorithm. The main objective is to reduce hotspots and to distribute temperature evenly accross a chip while minimizing its area. They define a transfer thermal resistance model and a slicing floorplan approach is studied represented by Polish expression. The genetic algorithm proposed aims to minimize the area needed in the first place. Then the algorithm is re-run only with the individuals that satisfy a good dead space ratio.

Floorplanning techniques evolve as the considered physical constraints change. For example, a common technique in VLSI is to insert flip flops to prevent global wire delay from becoming nonlinear, enabling deeper pipelines and higher clock frequency. Therefore, reducing wire delay has become crucial, hence floorplanning algorithms must take it into account. In the paper "Thermal-aware 3D Microarchitectural Floorplanning", Ekpanyapong et al. present a floorplanning algorithm that takes into consideration both thermal issues and profile weighted wire length using mathematical programming. The goal is to minimize the maximum temperature among all blocks and the overall execution time of a given processor. In this paper, the floorplanning problem is presented as a MILP problem. [18] uses a simulated annealing algorithm and an interconnect model to achieve thermal optimization. These works have a major restriction since they do not consider multiple objective factors in the optimization problem, as opposed to our work. Other works [25] have tackled the problem of thermal-aware floorplanning with geometric programming but, in this case, the area of the chip is not considered constant.

There are other approaches used to study the floorplanning problem, for example Guo and Takahashi present a genetic algorithm for the VLSI floorplanning problem using the ordered tree (O-tree) representation (see [16]). This representation covers all optimal floorplans and has a small search space. Once again, the goal of the genetic algorithm is to minimize the global area and the interconnection cost between the different modules. This representation allows to perform operations to the different floorplans with a reduced computational cost.

In our work, we propose to use application profiling techniques to guide the floorplanner. A work by [26] shows that the power profile does not have major effect on the leakage power as long as the total power remains same. However, they do not consider the effect of power profile on temperature variation across different applications, especially the peak temperature of the blocks. Only a recent work [30] incorporates multiple power profiles in a thermal-aware floorplanner. However, this work is not devoted to MPSoC and could not be easily extended to 3D multi-processor stacks, where most traditional thermal-floorplanner fail to find an optimal solution.

1.3 Contributions

The work presented in this thesis makes the following contributions:

- We adapt the ParMiBench suite to be run on bare machines and simulated with OVPsim to avoid the power overhead caused by an operating system.

- The dynamic profiles of different real world applications are retrieved from simulations of manycore heterogeneous architectures with OVPsim. These profiles are used to guide a thermal-aware floorplanner.

- We use a genetic algorithm capable of proposing thermally-optimized floorplans of architectures composed of 30, 6 and 128 cores. Previous research shows that MILP based techniques are unable to propose solutions in the latter case. The design of this floorplanner is based on a genetic algorithm capable of obtaining optimal solutions, in a short time, for a large number of integrated processors and layers and with minimal overhead.

- We propose for a first time an efficient thermal-aware 3D floorplanner for heterogeneous architectures of MPSoCs that uses as input the power traces obtained during an application power profiling phase.

- We obtain different floorplans by targeting different thermal objectives and we evaluate them with real power values retrieved the simulation of 6 different benchmarks.

- We show that considering the worst power consumption does not lead to optimal floorplans.

Chapter 2

Manycore and Heterogeneous architectures

Nowadays, manycore architectures are a current trend in research. Figure 2.1 shows the different strategies followed to improve the computers efficiency while respecting power and temperature constraints. More and more the different manufacturers tend to design multicore architectures. It is predicted that this architectures will be heterogeneous with special purpose components. This strategy allows to decrease the operating frequency and design platforms with a better thermal response.

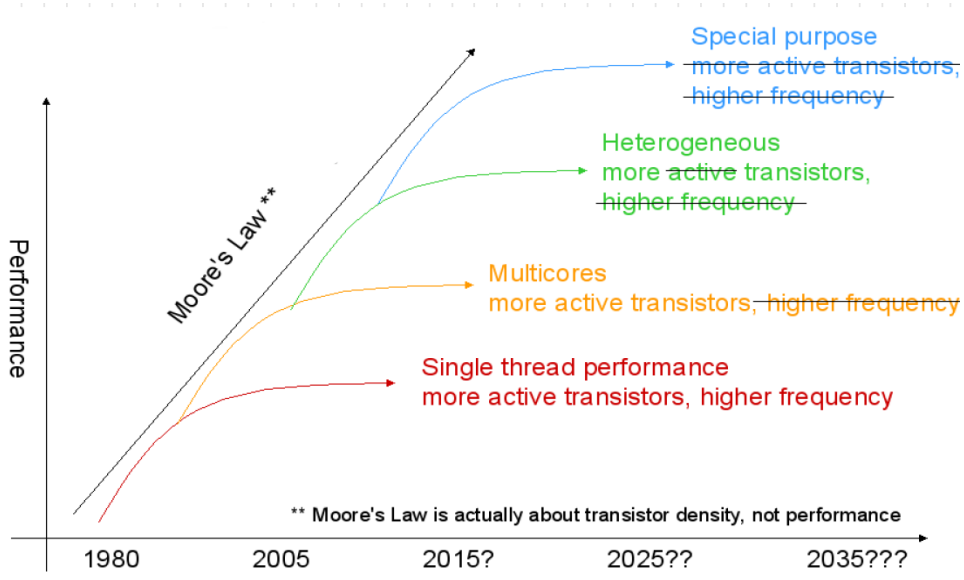


Figure 2.1: *Prediction of Future Trends in Computer Design*

2.1 Manycore architectures

A manycore processor is one in which the number of cores is large enough that traditional multiprocessor techniques are no longer efficient. It is due to issues with congestion in supplying instructions and data to the many processors. The manycore threshold is said to be in the range of several tens of cores; above this threshold network on chip technology is supposed to be advantageous.

Nowadays there are already several multicore and manycore architectures going from 12 to 100 cores architectures. Some of them are already available while others are still in a development phase and correspond to different approaches to multiprocessor architectures. For example, AMD has recently the "Magny-Cours" series, Oracle SPARC T3 (Niagara-3) is widely used in the server segment, Tilera is investing in research to produce the TILEGX chip, finally Intel's Many Integrated Core Architecture (MIC) is claimed to set the trend in design for the next years. In this section we study the Intel's Single-Chip Cloud Computer architecture and present the architectures studied in this work.

2.1.1 Intel's Single-Chip Cloud Computer

The Single-Chip Cloud Computer initiative is a research stream of Intel's Many Integrated Core (Intel MIC) project. The first Intel MIC products will target applications in High Performance Computing (HPC), Workstation, and Data Center segments that use highly parallel processing. The architecture utilizes a high degree of parallelism in smaller, lower power, and single threaded performance Intel processor cores, to deliver higher performance on highly parallel applications.

The SCC is the second generation processor design that resulted from Intel's Tera-Scale research. It has 24 tiles and two cores per tile. Each core has L1 and L2 caches. The L1 caches are on the core; the L2 caches are on the tile next to the core. Each core has a 16KB L1 instruction cache and a 16KB L1 data cache. Each core's L2 cache is 256KB. The SCC core is a full IA P54C core and hence can support the compilers and OS technology required for full application programming.

Users do not have to run a Linux image on the cores. Running Linux on the cores is the most common configuration, but not the only one as some users may be interested in one of the research operating systems being developed for many-core systems. In our case, we propose architectures inspired in this model. This architecture allows to deploy several applications in parallel, each one executed by a different group of processors. In this work, we emulate possible task distributions that could be found in the real platforms. In our case, we simulate the benchmarks on bare machines, therefore no operating systems are used.

2.2 Heterogeneous architectures

In general, an heterogeneous computing platform consists of processors with different instruction set architectures (ISA). But this definition is usually expanded, in fact, any electronic system that uses different types of computational units is considered heterogeneous. These computational units can be one of the following:

- a general-purpose processor (GPP)
- a co-processor
- a special-purpose processor:
  - digital signal processor (DSP)
  - graphics processing unit (GPU)
- custom acceleration logic:
  - application-specific integrated circuit (ASIC)
  - field-programmable gate array (FPGA)

Heterogeneous architectures are more and more demanded in computing systems. It is mainly due to the increasing need for high-performance and low power systems oriented to run audio, video control and network applications. The level of heterogeneity in modern computing systems gradually rises as increases the available chip area caused by the scaling of fabrication technologies. These systems present new challenges not found in typical homogeneous systems. The presence of multiple processing elements raises all the issues involved with homogeneous parallel processing systems, while the level of heterogeneity in the system can introduce non-uniformity in system development and programming practices. The most important problems that need to be solved when dealing with heterogeneous architectures are related to:

- Instruction Set Architecture (ISA): computing elements may have different instruction set architectures, leading to binary incompatibility
- Application Binary Interface (ABI): computing elements may interpret memory in different ways. This may include endianness, calling convention and memory layout. It depends both on the architecture and the compiler being used
- Application Programming Interface (API): libraries and operating systems services may not be uniformly available to all computing elements
- Low-Level Implementation of Language Features: language features such as functions and threads are often implemented using function pointers, a mechanism which requires additional translation or abstraction when used in heterogeneous environments

- Memory Interface and Hierarchy: computing elements may have different cache structures, cache coherency protocols, and memory access may be uniform or non-uniform (NUMA). Differences can also be found in the ability to read arbitrary data lengths as some processors/units can only perform byte, word, or burst accesses

- Interconnect: computing elements may have differing types of interconnections aside from basic memory/bus interfaces. This may include dedicated network interfaces, Direct memory access (DMA) devices, mailboxes, FIFOs, scratchpad memories, etc.

Heterogeneous platforms often require the use of multiple compilers to target the different types of computing elements found in such platforms. This results in a more complicated development process compared to homogeneous systems, as multiple compilers and linkers must be used together in a cohesive way to properly target an heterogeneous platform. Interpretive techniques can be used to hide heterogeneity, but the cost (overhead) of interpretation often requires the use of just-in-time compilation mechanisms that result in a more complex run-time system that may be unsuitable in embedded, or real-time scenarios.

Many processors now include built-in logic for interfacing with other devices (SATA, PCI, Ethernet, RFID, Radios, UARTs, and Memory Controllers), as well as programmable functional units and hardware accelerators (GPUs, Encryption Co-processors, programmable network processors, A/V encoders/decoders, etc.). Some real world examples are Toshiba's Spurs Engine used in Sony's Playstation 3, IBM's Cell research project and the recently released NVIDIA Tegra chip.

2.3 Proposed architectures

We have chosen the OVPsim simulator to achieve our experiments. OVPsim is a multiprocessor platform emulator that uses dynamic binary translation technology to achieve high simulation speeds. This simulator provides public API's allowing users to create their own processor, peripheral and platform models. In our case, we have used available processor models and have designed three different platforms. In this chapter, we explain briefly the API's provided by OVPsim, focusing on the ICM API which is the one that we have used, we also explain the core models used and finally, we give a detailed description of the designed architectures.

2.3.1 OVPsim API's

The models simulated with OVP are created using C/C++ API's. There are three main API's: ICM, VMI, BHM/PPM.

ICM The ICM API is used for controlling, connecting, and observing platforms. This API can be called from C, C++, or SystemC. The platform provides the basic structure of the design and creates, connects, and configures the components. The platform also specifies

the address mapping, and software that is loaded on the processors. It is very easy with ICM to specify very complex and complete platforms composed of:

- different processors
- local and shared memories
- caches
- bus bridges
- peripherals and all their complex address maps
- interrupts and operating systems
- application software

In our case we use only a few functions provided by the ICM API. A detailed explanation of the method followed to build the proposed architectures can be found in 2.3.4. In fact, a simple program that runs a given platform can be made using mainly five calls from the ICM API:

- *icmInit:* initializes the simulation environment prior to a simulation run: it should always be the first ICM routine called in any application. It specifies attributes to control some aspects of the simulation to be performed
- *icmNewProcessor:* used to create a new processor instance
- *icmLoadProcessorMemory:* Once a processor has been instantiated by *icmNewProcessor*, this routine is used to load an object file into the processor memory. Accepted formats are ELF and TI-COFF
- *icmSimulatePlatform:* used to run the simulation of the processor and program, for a specified duration
- *icmTerminate:* At the end of the simulation, this function should be called to perform cleanup and delete all allocated simulation data structures

VMI For processor modelling there is the VMI API. These API functions provide the ability to easily describe the behavior of the processor. A processor model written in C using the VMI decodes the target instruction to be simulated and translates it to native x86 instructions executed on the PC. VMI can be used for modelling 8, 16, 32, and 64 bit architectures. There is an interception mechanism enabling emulation of calls to functions in the application runtime libraries (such as write, fstat etc.) without requiring modification of either the processor model or the simulated application. We are not giving more details about this API as it has not been used in this work.

PPM and BHM Behavioral components, peripherals, and the overall environment are modelled using C code and calls to these two API's. Underlying these API's is an event based scheduling mechanism to enable modelling of time, events and concurrency. Peripheral models provide callbacks that are called when the application software running on processors modelled in the platform access memory locations where the peripheral is enabled. Adding callbacks across memory regions allows memory watchpoints. A callback is executed whenever there is either a read or write access to a specified range of memory addresses. With the given API, the callbacks are created using *icmAddReadCallback* and *icmAddWriteCallback* functions. We use these memory callbacks to count memory accesses.

Example

The following code adds a read watchpoint to the address range **0x01000000:0x01000fff**:

icmAddReadCallback(processor,0x01000000,0x01000fff,bufferReadCallBack,0)

The following code adds a write watchpoint to the address range **0x01000000:0x01000fff**:

icmAddWriteCallback(processor,0x01000000,0x01000fff,bufferWriteCallBack,0)

These watchpoints allow the monitoring of memory access behavior of a processor as it runs an application.

2.3.2 Core models

Within OVP there are several different model categories. These models are provided as both pre-compiled object code and as source files. Currently there are processor models of ARM (processors using the ARMv4, ARMv5, ARMv6, ARMv7 instruction sets), MIPS (processors using the MIPS32 and microMIPS instruction sets), ARC600/ARC700, NEC v850, PowerPC and OpenRisc families. There are also models of many different types of system components including RAM, ROM, cache and bridge. There are also peripheral models including DMA, UART, and FIFO. There are also models of several different prebuilt platforms including software like ucLinux to run on them. One of the main uses of the OVP simulation infrastructure is the ability to create and simulate models, either from scratch, or by using one of the open source models as a starting point. In our case, we use a SPARC model, an ARM model and a PowerPC model.

SPARC

SPARC (from Scalable Processor Architecture) is a RISC instruction set architecture (ISA) developed by Sun Microsystems and introduced in mid-1987. The "Scalable" in SPARC comes from the fact that the SPARC specification allows implementations to scale from embedded processors up through large server processors, all sharing the same instruction

set. One of the architectural parameters that can scale is the number of implemented register windows; the specification allows from 3 to 32 windows to be implemented, so the implementation can choose to implement all 32 to provide maximum call stack efficiency, or to implement only 3 to reduce context switching time or to implement some number between them. The endianness of the 32-bit SPARC V8 architecture is purely big-endian. The 64-bit SPARC V9 architecture uses big-endian instructions, but can access data in either big-endian or little-endian byte order, chosen either at the application instruction (load/store) level or at the memory page level (via an MMU setting).

We decide to include SPARC cores in our platforms because it is a commonly used processor in architectures targeting the server segment. On the other hand, these processors have a high power consumption. Hence, thermal-aware design is mandatory when working with these models. As we want to study architectures comparable to the ones predicted to be released in the short term, we need to include processors with different computing power as well as different power consumptions. In this case, the SPARC cores are the most powerful ones but also the hottest elements of our platforms. Therefore, their placement is crucial to obtain chip configurations with an acceptable thermal behaviour.

ARM

ARM is a 32-bit RISC instruction set architecture developed by ARM Holdings. It is known as the Advanced RISC Machine. The ARM architecture is the most widely used 32-bit ISA in terms of numbers produced. The relative simplicity of ARM processors makes them suitable for low power applications. As a result, they have become dominant in the mobile and embedded electronics market, as relatively low-cost, small microprocessors and microcontrollers. ARM processors are developed by ARM and by ARM licensees. Prominent ARM processor families developed by ARM Holdings include the ARM7, ARM9, ARM11 and Cortex. In our case, we do not work with a specific core model but with the Instruction Set Architecture (ISA).

This architecture represents the paradigm in embedded design processors. Nowadays it is the most produced architecture and its computing power is increasing in every new generation of processors. We include ARM cores in our architectures because we find interesting to mix processor models with very different power consumption. This way, we evaluate the floorplanner's ability to strategically place the different cores to avoid hotspots. The ARM cores are also much smaller than the SPARC or the PPC ones. It is also interesting to test how the floorplanner deals with the different topological constraints of the different processor models. In our case, the ARM cores will the medium ones in terms of computing power.

PowerPC

PowerPC (short for Performance Optimization With Enhanced RISC - Performance Computing, sometimes abbreviated as PPC) is a RISC architecture created by the 1991 Apple-IBM-Motorola alliance, known as AIM. PowerPC has been renamed Power ISA since 2006 but lives on as a legacy trademark for some implementations of Power Architecture based processors. This architecture was originally intended for personal computers, PowerPC CPUs have since become popular as embedded and high-performance processors. PowerPC was known for being used by Apple's Macintosh lines from 1994 to 2006 but its use in video game consoles and embedded applications far exceeded Apple's use. PowerPC is largely based on IBM's earlier POWER architecture, and retains a high level of compatibility with it. In fact, the architectures have remained close enough that the same programs and operating systems will run on both if some care is taken in preparation. Newer chips in the POWER series implement the full PowerPC instruction set. In our case, we do not work with a specific core model but with the Instruction Set Architecture (ISA).

The choice of the PPC cores is justified with the same reasons that led us to study the ARM cores. We include three different processors with different computing power and different topological constraints. In this case, the considered PPC model (see 4.2.2) is also a low power oriented processor but it is much bigger than the included ARM. The main reason to include another low power processor is that these two cores will be best suited for different applications. This means that optimized compilers or operating systems may take advantage of this fact and increase both efficacy and efficiency of a given platform. Moreover, including low power processors in our platforms allows us to increase the parallelism of the programs running in our platforms without increasing drastically the temperature of the chip.

2.3.3 Simulated manycore heterogeneous architectures

OVPsim allows the simulation a a variety of platforms. In our case we are interested in the study of manycore heterogeneous architectures. The main elements composing our architectures are processors and memories. In fact we will study three different cases that differ from one to another in the number of cores. In all of the cases, there is a shared memory common to all the processors (used for the inter-processor communication) and a local memory for each of the cores. The proposed platforms are inspired in the Intel's Single-Chip Cloud Computer project but we include low power cores that might lead to colder chips. This is a significative change and implies extra difficulties such as different Instruction Set Architectures and endianness among others.

The platforms considered are heterogeneous, composed of SPARC, ARM and PowerPC processors. We are interested in discovering whether this kind of architectures are feasible solutions when we scale up the number of cores. We design the different platforms to compare the thermal behavior of architectures with different degrees of heterogeneity.

In the smallest architecture there are 30 cores with the following distribution: 20 SPARC, 5 ARM and 5 PPC. This case corresponds to a multicore platform with a high proportion of SPARC cores. Therefore the floorplanner does not have much freedom to place the hottest elements as far away from each other as possible. The most promising strategy is to place the SPARC cores in the borders of the chip and in the outer layers. This case corresponds to architectures that target the server or HPC segments including special purpose units (the low power processors in our case) that allow to reduce the heat produced by the chip.

The medium platform is composed of 22 SPARC, 22 ARM and 22 PPC adding up a total of 66 cores. In this case there is an homogeneous number of the different core models. It is interesting to see how the floorplanner deals with the hottest elements (SPARC) with a higher degree of freedom. There are already platforms designed within this range of cores. The higher the number of cores, the highest is the need to include low power processors. In fact, homogeneous chips with a high number of SPARC cores present dramatic hotspots that can only be solved by adding extra area to separate the cores as much as possible.

Finally, in the last scenario, there are 129 cores: 43 SPARC, 43 ARM and 43 PPC . This case correspond to a scaled up version of the 66 processors architecture. We consider the study of this case very important as it corresponds to the predicted trend in computer design in the short term. There are already projects like Intel's SCC due in the next years that implement a similar number of cores. The higher the number of cores, the harder is to find solutions for the floorplanner. It is due to the fact that the search space increases exponentially with the number of considered elements. A problem of such a dimension forces us to research new floorplanning techniques and algorithms. In fact, for problems of this size, traditional thermal-aware techniques fail to return solutions with an acceptable thermal response in a reduced response time. Therefore considering this case represents a challenge and finding acceptable solutions is one of the contributions of this work.

In our platforms, each processor has its own local memory. To allow communication between processors, we adopt a shared memory strategy. The size of the local memories must be small, as manycore architectures with big local memories are not feasible. On the other hand the size of the shared memory depends on the architecture as more processors need a larger memory space to communicate and share data. The exact size of the memories considered is fixed later on with profiling techniques (see section 4.2.1).

2.3.4 Designing manycore heterogeneous architectures with OVPsim

In this subsection, we explain in detail the procedure followed to build the proposed architectures with OVPsim. We show the main steps and the different calls to the OVPsim API.

1. In the first place we must define the macros for each memory callback. For example, the memory callbacks of the shared memory are defined as follows:
   - *static* $ICM\_MEM\_WRITE\_FN(watchWriteCBShared)\ \{writesShared{+}{+};\}$
   - *static* $ICM\_MEM\_WATCH\_FN(watchReadCBShared)\ \{readsShared{+}{+};\}$
2. we initialize OVPsim, enabling verbose mode to get statistics at end of execution with the following function call:
   - *icmInit(*$ICM\_VERBOSE|..., NULL, 0$*);*
3. we create an array of pointers to processor instances:
   - *icmProcessorP processor*$[num\_processors]$*;*
4. we create the shared memory:
   - *icmMemoryP shared = icmNewMemory("shared",* ICM_PRIV_RWX*, 0x1fffffff);*
5. we create MMC to perform endian swap needed to hide the endianness problem:
   - *icmAttrListP icmAttrListMMC = icmNewAttrList();*
   - *const char * string_mmc =icmGetVlnvString ("mmc","endianSwap",...);*
   - *icmMmcP icmMmcP_mmc = icmNewMMC("swap1",string_mmc,...);*
6. we create bus from the MMC to the shared memory for the PPC's:
   - *icmBusP busMmcIn = icmNewBus("busMmcIn", 32);*
   - *icmBusP busMmcOut = icmNewBus("busMmcOut", 32);*
7. MMC input/output bus:
   - *icmConnectMMCBus(icmMmcP_mmc, busMmcIn,"sp1",0);*
   - *icmConnectMMCBus(icmMmcP_mmc, busMmcOut,"mp1",1);*
8. MMC output bus to shared memory:
   - *icmConnectMemoryToBus(busMmcOut, "mpswap", shared, 0xa0000000);*
9. create each of the processor models, we show the case of the ARM model:
   - *const char *armModel = icmGetVlnvString("arm.ovpworld.org", "processor","arm",);*
   - *const char *armSemihost = icmGetVlnvString("arm.ovpworld.org","armNewlib");*
10. We set the attribute list for the ARM:
    - *icmAttrListP icmAttr = icmNewAttrList();*

- *icmAddStringAttr(icmAttr, "variant", "Cortex-A9");*

11. for each of the processors:

    - *processor[i] = icmNewProcessor(*
      cpuName, // CPU name
      "arm", // CPU type
      i, // CPU cpuId
      0, // CPU model flags
      32, // address bits
      armModel, // model file
      "modelAttrs", // morpher attributes
      SIM_ATTRS, // attributes
      icmAttr, // user-defined attributes
      armSemihost, // semi-hosting file
      "modelAttrs" // semi-hosting attributes);
    - create one bus for each processor instantiation:
      icmBusP bus = icmNewBus(busName, 32);
    - connect the processor onto the bus:
      icmConnectProcessorBusses(processor[i], bus, bus);
    - create memories: the ARM processor toolchain sites code in lower memory and stack in higher memory, so we use two memories as a consequence of the default linker script used:
      icmMemoryP locala = icmNewMemory(localaName, ICM_PRIV_RWX, 0x9fffffff);
      icmMemoryP localb =icmNewMemory(localbName,ICM_PRIV_RWX,0x0fffffff);
    - connect local memories onto individual processor bus:
      icmConnectMemoryToBus(bus, "mp1", locala, 0x00000000);
      icmConnectMemoryToBus(bus, "mp1", localb, 0xf0000000);
    - connect the shared memory onto the local bus, this makes it available to all processors at the specified address in this case, but it could be at any address in each processors address map:
      icmConnectMemoryToBus(bus, mpName, shared, 0xa0000000);
    - load an executable file to the processor local memory, the file loaded depends on the processor id:
      icmLoadProcessorMemory(processor[i], "bitcntsParallelArm.ARM7.elf",False,False, True))

12. the same method is used to declare and link the other processor models

It is important to note that we connect all the processor buses onto the shared memory in the same address range, this way the shared memory is available to all the processors at the same address range. An issue that must be solved when dealing with this kind of

heterogeneous architectures is the endianness. In fact, the chosen processor models have different endianness which makes communication between them impossible. The ARM model is only available in an little-endian architecture while the PPC model is only available in a big-endian architecture. On the other hand, the SPARC model can access data in either big-endian or little-endian byte order. In the case of the PowerPC processors, we use endian swappers between the processors buses and the shared memory. We must also use bridges to make clear which address range must be connected to the local memory and which address range must be connected to the endian swapper. This way, the endianness problem can be ignored from now on. In fact, the software running in this architectures does not need to care about this problem as it is solved with hardware components.

In figure 2.2, we show a representation of an heterogeneous architecture created following the ideas just explained. This heterogeneous architecture is composed of:

- nine processors: three SPARC, three ARM and three PowerPC
- their corresponding local memories divided in two regions: the code is located in lower memory and stack is located in higher memory
- a shared memory
- an endian swapper, only used by the PowerPC processors
- bus bridges for the PowerPC processors

Another issue is related to the toolchains available that allow us to cross-compile a given code to be executed in different targets. In fact, these toolchains have pre-established memory addresses for the code, data, etc. Therefore we have to map the processor buses to these memory regions if we want to make an efficient use of the memories.

2.3.5 Multiprocessor Scheduling Algorithm

With the OVPsim software, it is possible to simulate multiprocessor platforms. In fact, any number of processors can be instantiated within an ICM platform. Shared memory resources and callbacks on mapped memory regions are used to allow communication between them.

The provided function *icmSimulatePlatform* implements the following multiprocessor scheduling algorithm:

1. Simulation time is broken into time slices. By default, each time slice is 0.001 seconds (one millisecond)
2. The simulator selects the first processor and simulates it for one time slice. First, the number of instructions that should be executed by that processor in a time slice is computed, and then the processor is simulated for that number of instructions. The number of instructions in a time slice is:
   (processor nominal MIPS rate) $\times$ *1e6* $\times$ *(time slice duration)*

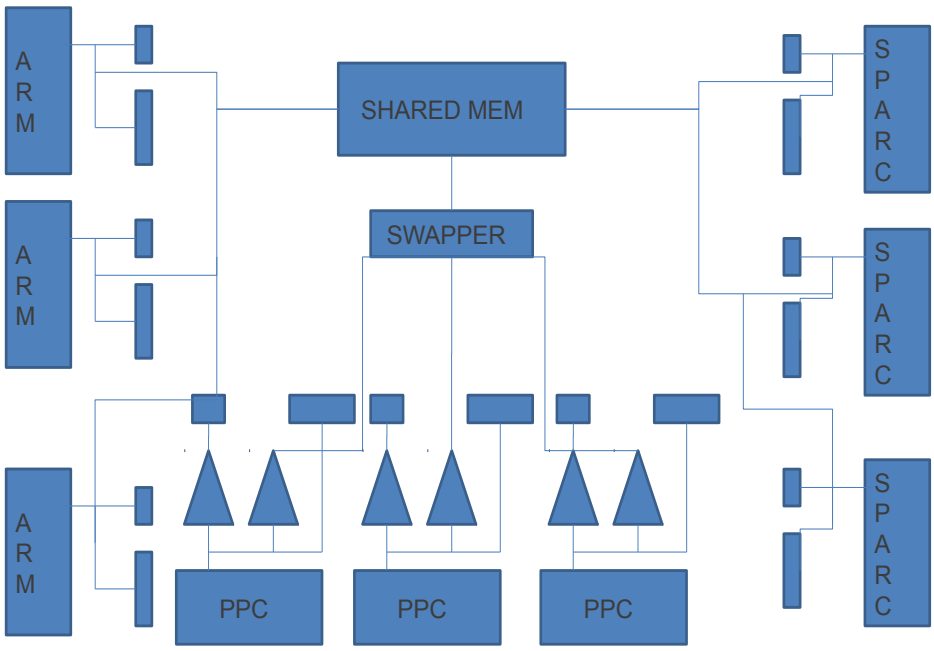


Figure 2.2: *Example of heterogeneous architecture*

3. When the first processor has simulated the corresponding instructions, it is suspended and the next processor is simulated for the time slice

4. When all the processors have simulated the time slice, simulated time is moved on and the next slice is started

This algorithm is an approximation designed to give realistic simulation results with high simulator performance: the simulator is not designed to be cycle accurate. The simulation algorithm is configurable by changing the time slice or changing the processor nominal MIPS rate. OVPsim allows us to set these parameters:

- *Bool icmSetSimulationTimeSlice(icmTime newSliceSize)*. Shorter time slices may approximate real system behavior more closely, but degrade simulator performance

- The nominal MIPS rate for each processor can be set with an attribute.

To obtain realistic power profiles of heterogeneous platforms, we have chosen three processors with a different computing power, which is a common situation when dealing with this kind of architectures. Therefore the MIPS rate is set accordingly to the computing power of each processor. As a result the processor with a higher MIPS rate is the SPARC core (120), followed by the ARM (80) and the PPC (60). In fact, we are specially interested

in understanding the effect of synchronization and communication in the temperature of the chip. Figure 2.3 shows a typical power dissipation pattern of three different processors working together. We can see clearly how the activity of the SPARC core changes periodically over time, waiting for slower processors. This kind of behavior leads us to think that considering always the highest power consumption for each element may lead to an overestimation of the dissipated power and temperature of the chip. Therefore, taking into account the data retrieved in the proposed power profiling phase will lead to better floorplans.

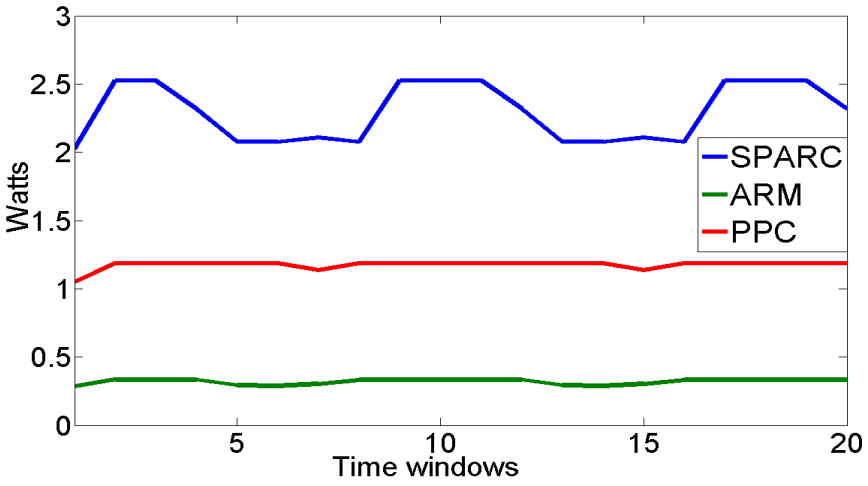


Figure 2.3: *Common power consumption pattern caused by synchronization*

OVPsim also allows us to write our custom scheduling algorithm in case the standard multiprocessor scheduling algorithm does not do what it is required. A custom algorithm can be built around calls to *icmSimulate* for each processor. This function will simulate a specified processor for an exact number of instructions. In our case, we need to retrieve profiling data that is not provided by OVPsim's standard statistical engine. Therefore, we need to implement our own multiprocessor scheduling algorithm. In particular, the data we need to obtain is:

- for each simulation window:
  - for each processor:
    - the amount of executed instructions
    - the amount of idle cycles
  - for each memory (local and shared):
    - the amount of read accesses
    - the amount of write accesses

The algorithm proposed is a round robin based scheduling strategy. The detail of this algorithm can be seen in the appendix B. The benchmarks chosen for this work are presented in Chapter 4, as well as the task distributions decided for the different platforms.

Chapter 3

An Evolutionary Algorithm for the 3D Floorplanning problem

Floorplanning has been proved to be a crucial step in VLSI (Very Large Scale Integration) physical design. Traditionally, its main objective has been to minimize the total area required to place all of the functional blocks on a chip. The floorplanning problem is a generalization of the quadratic assignment problem, which is an NP-hard problem. The different approaches to this problem can be divided into three general cases: constructive, knowledge-based, and iterative. The constructive approach starts from a seed module, and adds modules to the floorplan until all modules have been selected. The knowledge-based approach uses a database with expert knowledge to guide the floorplan development. The iterative approach starts from an initial floorplan which undergoes a series of transformations until the optimization goal is reached. The iterative class includes force directed relaxation methods, simulated annealing, and evolutionary algorithms. In our case, we are interested in the last approach.

Most of the algorithms presented for the 3D thermal aware floorplanning problem are based on a Mixed Integer Linear Program (MILP) [19], [24], Simulated Annealing (SA) [11], [19] or Genetic Algorithm (GA) [31]. MILP has proven to be an efficient solution. However, when MILP is used for thermal aware floorplanning, the (linear) thermal model must be added to the topological relations and the resultant algorithm becomes too complex [14], specially as the problem size (number of cores, in our case) increases. For example, techniques based on a MILP formulation of the problem fail to return solutions in the 129 cores scenario. Regarding SA and GA, the main problem is based on the representation of the solution. Some common representations are polish notation [5], combined bucket array [11] and O-tree [31]. Most of these representations do not perform well, because they were initially developed to reduce area. Moreover, they place the different elements right next to each other and do not let any space between them. This is due to the fact that they were initially conceived to reduce area, not to satisfy thermal constraints. In the thermal aware floorplanning problem, hottest elements must be placed as far as possible in the 3D IC. In this work, we have developed a straightforward Multi-Objective Evolutionary Algorithm

(MOEA) based on NSGA-II [13], which tries to minimize maximum temperature and total wire length while fulfilling all the topological constraints.

In this chapter, we present an introduction to Multi-Objective Evolutionary Algorithms, focusing on the NSGA-II and the SPEA2 algorithms. We also explain the most common representations of the floorplanning problem used in the literature and finally, we explain the thermal-aware genetic algorithm proposed in this work.

3.1 Multi-Objective Evolutionary Algorithms (MOEA)

Evolutionary algorithms form a subset of evolutionary computation which in turn is a subfield of artificial intelligence that involves combinatorial optimization problems. Evolutionary computation uses iterative progress, such as growth or development in a population. This population is iterativetily evolved in a guided pseudo-random search to achieve the desired end. The processes followed are inspired by biological mechanisms of evolution such as reproduction, mutation, natural selection and survival of the fittest. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the adaptation to the environment where the solutions live. The evolution of the population takes place after the repeated application of these operators. Figure 3.1 shows the typical flow of an evolutionary algorithm. Two main forces form the basis of evolutionary systems:

- Recombination and Mutation create the diversity
- Selection acts as a refinement of the population

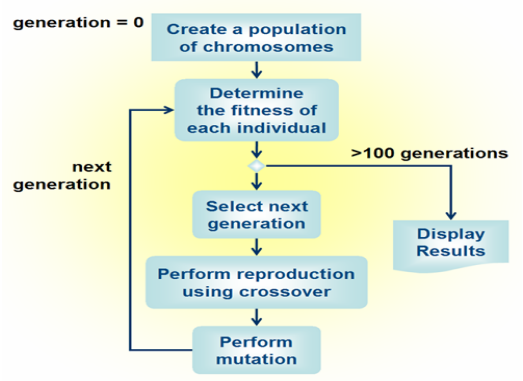


Figure 3.1: *EA Schema*

Many aspects of evolutionary processes are stochastic. Changed pieces of information due to recombination and mutation are randomly chosen. On the other hand, selection operators can be either deterministic, or stochastic. In the latter case, individuals with a higher fitness have a higher chance to be selected than individuals with a lower fitness,

but typically even the weak individuals have a chance to become a parent or to survive. This chance allows to incorporate to the population good characteristics of worst adapted individuals, and eventually to obtain optimal solutions combining these features with other better individuals.

MOEAs are stochastic optimization heuristics where the exploration of the solution space of a certain problem is carried out by imitating the population genetics stated in Darwin's theory of evolution. Selection, crossover and mutation operators, derived directly from natural evolution mechanisms, are applied to a population of solutions, thus favoring the birth and survival of the best solutions. MOEAs have been successfully applied to many NP-hard combinatorial optimization problems and work by encoding potential solutions (individuals) to a problem by strings (chromosomes), and by combining their codes and, hence, their properties. In order to apply MOEAs to a problem, a genetic representation of each individual has first to be found. Furthermore, an initial population has to be created, as well as defining a cost function to measure the fitness of each solution.

In general we can consider, two main types of multi-objective evolutionary algorithms:

1. The algorithms that do not incorporate the concept of optimal Pareto selection mechanism on the evolutionary algorithm (using linear aggregative functions).

2. The algorithms that hierarchically organize the population according to whether an individual is not dominated or not (using the concept of Pareto optimal). Examples: MOGA, NSGA, NPGA, etc.

Historically we consider that there were two generations of multi-objective evolutionary algorithms:

1. First Generation: Characterized by the use of hierarchy based on Pareto dominance and niches. Relatively simple algorithms.

2. Second Generation: It introduces the concept of elitism in two main ways: using selection and using a secondary population

On 2001 first-generation algorithms started to fall into disuse (MOGA,NSGA, NPGA, and VEGA). Since then evolutionary algorithms using multi-objective elitism are viewed as the state of art in the area (SPEA, SPEA2, NSGA-II, GMMOs, GMMs-II, PAES, PESA, PESAII, etc.). See [10, 15] and [32].

The main advantage of evolutionary algorithms, when applied to solve multi-objective optimization problems, is the fact that they typically optimize sets of solutions which allows to compute an approximation of the entire Pareto front in a single algorithm run. Figure 3.2 shows an approximated Pareto Set (squares). All the solutions of this front are the non dominated solutions found by the algorithm. On the other hand the triangles represent

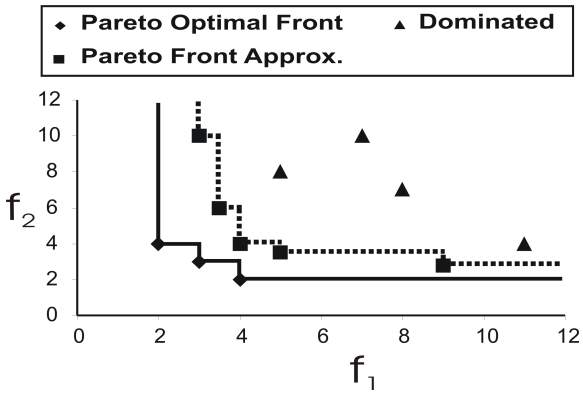


Figure 3.2: *Example of an approximated Pareto front*

dominated solutions. The Pareto Dominance concept is used as a criterion to rank and select the fittest solutions.

We present an overview of the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) and Strength Pareto Evolutionary Approach 2 (SPEA-2). In fact, these two methods have become standard approaches to deal with multi-objective optimization problems:

3.1.1 SPEA2

The Strength Pareto Evolutionary Algorithm (SPEA) algorithm was conceived as a way of integrating different MOEAs. SPEA uses an external non-dominated set which is in fact an archive containing non-dominated solutions previously found. At each generation, non-dominated individuals are copied to the external non-dominated set. For each individual in this external set, a strength value is computed. This strength is proportional to the number of solutions that it dominates. The external non-dominated set is the elitist mechanism adopted. In SPEA, the fitness of each member of the current population is computed according to the strengths of all the external non-dominated solutions that dominate it.

The Strength Pareto Evolutionary Algorithm 2 (SPEA2) has three main differences with respect to its predecessor:

1. It incorporates a fine-grained fitness assignment strategy which takes into account not only the individuals that dominate a given solution but also the number of individuals that this solution dominates.

2. It uses a nearest neighbor density estimation technique which guides the search more efficiently.

3. The elitist mechanism is improved: SPEA2 presents an enhanced archive truncation method that guarantees the preservation of boundary solutions.

3.1.2 NSGA-II

The Non-dominated Sorting Genetic Algorithm (NSGA) is based on several layers of classification of the individuals. Before selection is performed, the population is ranked on the basis of Pareto dominance: all non-dominated individuals are classified into one category. Then this group of classified individuals is ignored and another layer of non-dominated individuals is considered. The process continues until all individuals in the population are classified. Since individuals in the first front have the maximum fitness value, they always get more copies than the rest of the population. This allows to search for non-dominated regions, and results in convergence of the population towards such regions.
NSGA-II is a more efficient version of his predecessor. It uses elitism and a crowded comparison operator that keeps diversity without specifying any additional parameters. The elitist mechanism consists of combining the best parents with the best offspring obtained. We are specially interested in this algorithm as the floorplanner proposed in this work in based on it. Figure 3.3 shows the flow of the NSGA-II:

1. Set t, the generation count, to 0
2. Generate the initial population $P(t)$ of μ individuals, each represented as a set of real vectors, (x_i,η_i), $i = 1, ..., \mu$. Both x_i and η_i contain N independent variables: $x_i = \{x_i(1), ..., x_i(N)\}$, $\eta_i = \{\eta_i(1), ..., \eta_i(N)\}$
3. Evaluate the objective vectors of all individuals in $P(t)$ by using the multi-objective function
4. Calculate the rankings and crowding distances of all individuals
   (a) Execute $DominanceChecking(P(t), C, S)$
   (b) Execute $NonDominatedSelection(P(t), C, S, V, \mu)$
5. While the termination condition is not satisfied
   (a) For i from 1 to $\mu/2$, select two parents $P^1_{parent_i1}$ and $P^2_{parent_i1}$ from $P(t)$ using the tournament selection method
   (b) For i from 1 to $\mu/2$, recombine $P^1_{parent_i1}$ and $P^2_{parent_i1}$ using single point crossover to produce two offspring stored in the temporary population P^2. The population P^2 contains μ individuals
   (c) Mutate individuals in P^2 to generate modified individuals stored in the temporary population P^3. For an individual $P^2_i = (x_i, \eta_i)$, where $i = 1, ..., \mu$, create a new individual $P^3_i = (x_i^{'}, \eta_i^{'})$
   (d) Evaluate the objective vectors of all individuals in P^3
   (e) Combine the parent population $P(t)$ with P^3 to generate a population P^4 containing 2μ individuals
   (f) Check the dominance of all individuals in P^4 by executing $DominanceChecking(P^4, C, S)$
   (g) Select μ individuals from P^4 and store them in the next population $P(t+1)$. The individuals are selected by executing $NonDominatedSelection(P^4, C, S, V, \mu)$
   (h) $t = t + 1$
6. Return the non-dominated individuals in the last population

Figure 3.3: *NSGA2 flow*

3.2 Common representations

In this section we explain the Slicing Structures and O-tree representations generally used to approach the floorplanning problem.

3.2.1 Slicing Structures

In "A Slicing Structure Representation for the Multi-layer Floorplan Layout Problem" [5], a genetic algorithm is used to solve the multiple layer floorplanning problem. Its main contribution is a three dimensional slicing structure representation. A floorplan is encoded in normalized Polish expressions where the symbols "H", "V", and "Z", represent horizontal, vertical, and lateral cuts respectively. The authors propose to break down the 3D slicing structure to evaluate a floorplan. They define a slicer algorithm which accepts a 3D floorplan and the maximum number of layers, and returns a slicing structure for each layer. For example, Figure 3.4 shows the binary tree for the Polish expression "3 1 6 8 Z H Z 2 7 Z V 5 4 H V" The three trees on the right side of Figure 3.4 show the same Polish expression divided into three layers. Figure 3.5 shows the floorplans obtained for the different layers.

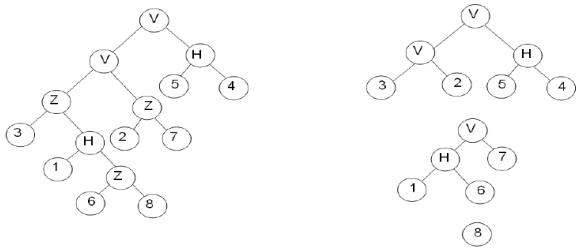


Figure 3.4: *The 3D floorplan tree for "3 1 6 8 Z H Z 2 7 Z V 5 4 H V" and its 2D layers*

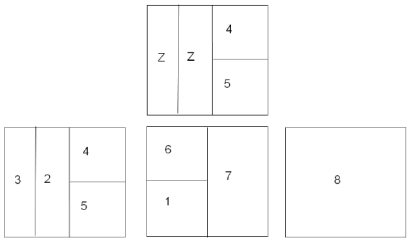


Figure 3.5: *The 3D slicing floorplan (top), and the three slicing floorplan layers (bottom). Structures marked "Z" in the 3D floorplan denotes subtrees branching upwards.*

With this representation, a genetic algorithm is used to minimize whether the overall total area or the balanced area. The advantage of this representation is that the crossover

and mutation operators are very fast, on the other hand, polish representations are not suitable for thermal-aware floorplanning. Nevertheless, slicing structures remain one of the most used representations for the floorplanning problem.

In "Thermal-Aware Floorplanning Using Genetic Algorithms"[21], a genetic algorithm based thermal-aware floorplanning framework is presented. The main objective is to reduce hotspots and to distribute temperature evenly across a chip while minimizing its area. They define a transfer thermal resistance R_{ij} of functional $block_i$ with respect to $block_j$ as the temperature rise at $block_i$ due to one unit of power dissipated at $block_j$. A slicing floorplan approach is studied represented by Polish expression. The genetic algorithm proposed aims to minimize the area needed in the first place. Then the algorithm is re-run only with the individuals that satisfy a good dead space ratio. In this genetic algorithm, the mutation operator consists in the rotation of a given solution.

3.2.2 O-trees

Tang and Sebastian present a genetic algorithm for the VLSI floorplanning problem using the ordered tree (O-tree) representation in [31]. This representation covers all optimal floorplans and has a small search space. The goal of the genetic algorithm is to minimize the global area and the interconnection cost between the different modules. In the O-tree representation, a floorplan of n modules is represented in a horizontal ordered tree of $(n+1)$ nodes, of which n nodes correspond to n modules $m_1, m_2, ..., m_n$ and one node corresponds to the left boundary of the floorplan. The left boundary is a dummy module with zero width placed at $x = 0$. In this representation, there is a directed edge from module m_i to module m_j if and only if $x_j = x_i + w_i$, where x_i is the x coordinate of the left-bottom position of m_i, x_j is the x coordinate of the left-bottom position of m_j, and w_i is the width of m_i. An ordered tree of n nodes can be encoded in a tuple (T, π), where T is a $2(n-1)$ bit string identifying the structure of the ordered tree and π is a permutation of the $n-1$ non-root nodes. For a horizontal O-tree, the tuple is obtained by DFS (Depth-First Search) traversing the non-root nodes and edges of the O-tree. When visiting a non-root node, we append it to π. When visiting an edge in descending direction we append an 0 to T and when visiting an edge in ascending direction we append a 1 to T. Figure 3.6 shows an example of an horizontal ordered tree representation. The same idea can be used to encode a vertical O-tree.

This representation allows to perform operations to the different floorplans with a reduced computational cost. We explain here the crossover and mutation operations proposed by Tang and Sebastian:

- Given two parents, both of which are O-trees, the crossover generates one child by recombining meaningful structural components from the two parents. It is observed that branches of an O-tree are meaningful structural components because a branch represents a potential compact placement for a given set of modules. Hence, the crossover uses branches of an O-tree as basic building blocks to generate an offspring.

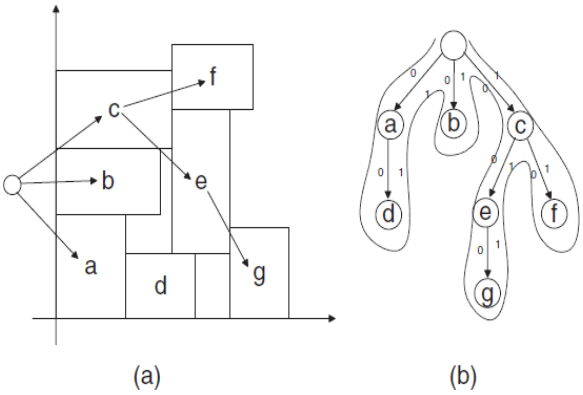
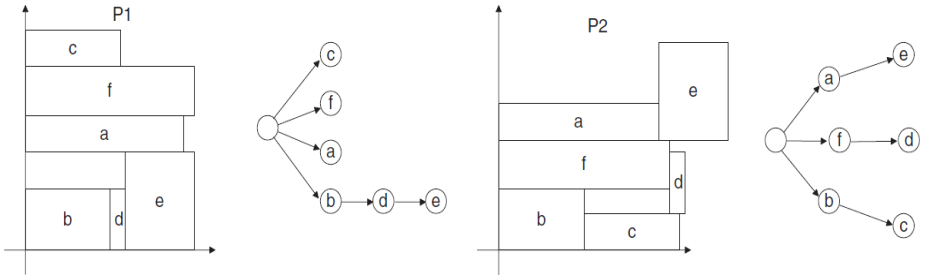
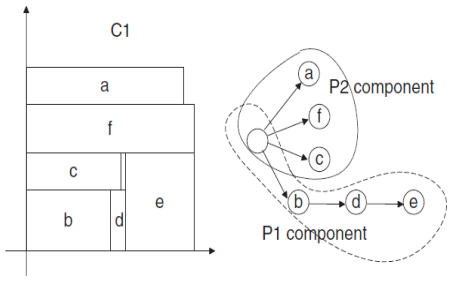


Figure 3.6: *Horizontal ordered tree encoded into* $(00110100011011, adbcegf)$

When generating an offspring $c1$ from two parents $p1$ and $p2$, the crossover randomly selects some branches from $p1$, duplicates them and puts them in $c1$. Then, the crossover operator takes a copy of $p2$ and removes the modules that have already been placed in $c1$ from it and adds it to $c1$. Figure 3.7 illustrates the basic idea behind the crossover operator 3.7(a) and 3.7(b) are two O-trees, $p1$ and $p2$ respectively, and 3.7(c) is the offspring generated by the crossover operator. The corresponding placements are shown on the left hand side of the figures.



(a) Parent P1 (b) Parent P2



(c) Offspring C1

Figure 3.7: *Parent 1 (P1), Parent 2 (P2) and the generated offspring (C1)*

- Given an individual, or an O-tree encoded in a tuple (T, π), the mutation randomly re-permutates the sequence of the modules. The mutation does not change the topology of the O-tree, but generates a different placement. Suppose that *(0011010001101,adbcegf)* is an initial individual, and that *abcdefg* is the randomly generated sequence of the module labels. The mutated individual is *(0011010001101,abcdefg)*. The mutation is illustrated in Figure 3.8. The computational complexity of this mutation is $O(n)$, where n is the number of modules.

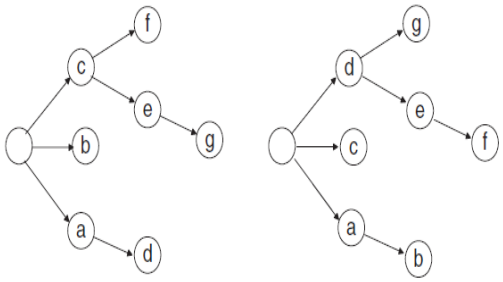


Figure 3.8: *Initial O-tree (a) and the mutated O-tree (b)*

In the previous examples, the different solutions are encoded with strings. These representations allow efficient implementations of the different operators involved in evolutionary algorithms. In our case, the individuals must correspond to feasible solutions all along the execution of the algorithm. This requires a validation step of the new solutions which translates into an extra computational cost. Moreover, these representations are not suited for thermal-aware floorplanning optimization. Hence, a new representation of the problem is needed to efficiently target our problem. In the next section we propose a thermal-aware floorplanner based on the NSGA-II.

3.3 Proposed Algorithm

The designed genetic operators presented in this section allow us to produce a population of thermal-aware floorplaning solutions by iteratively applying these operators to an initial population. The best individuals in the population converge to targeted solutions according to the metrics to be optimized.

3.3.1 Genetic representation and operators

A genetic representation of the design space of all possible floorplanning alternatives is needed to apply a MOEA correctly. Moreover, to be able to apply the NSGA-II optimization process and cover all possible inter-dependencies of the topological constraints, we must guarantee that all the chromosomes represent real and feasible solutions to the problem and ensure that the search space is covered in a continuous and optimal way. To this end, we propose the following representation:

- Every block i in the model $B_i(i = 1, 2, \ldots, n)$ is characterized by a width w_i, a height h_i and a length l_i while the design volume has a maximum width W, maximum height H, and maximum length L. We define the vector (x_i, y_i, z_i) as the geometrical location of block B_i, where $0 \leq x_i \leq L - l_i$, $0 \leq y_i \leq W - w_i$, $0 \leq z_i \leq L - h_i$. We use (x_i, y_i, z_i) to denote the left-bottom-back coordinate of block B_i while we assume that the coordinate of left-bottom-back corner of the resultant IC is $(0, 0, 0)$.

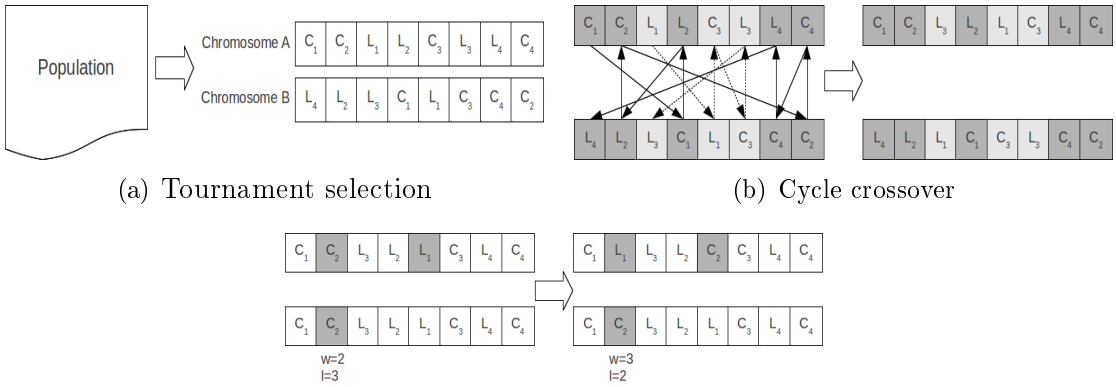
- we use a permutation encoding [9], where every chromosome is a string of labels, that represents a position in a sequence.

Figure 3.3.1 depicts the three genetic operators used in our MOEA on a floorplanning problem. A chromosome in Figure 3.3.1 is formed by 4 cores $C_i(i = 1, 2, 3, 4)$ and 4 memories $L_i(i = 1, 2, 3, 4)$. In every cycle of the optimization process (called generation) we follow the next process:

1. **Two chromosomes are selected by tournament**: we select two random chromosomes from the whole population and we select the best of these. This task is repeated twice to obtain two chromosomes (called parents, see Figure 3.9(a)).

2. We apply the **cycle crossover** (Figure 3.9(b)): starting with the first allele of chromosome A (C_1), we look at the allele at the same position in chromosome B. Next, we go to the position with the same allele in A, and add this allele to the cycle. Then, we repeat the previous step until we arrive at the first allele of A. Finally, we put the alleles of the cycle in the first child on the positions they have in the first parent, and take next cycle from second parent.

3. **Mutation** can be executed in two different ways, both with the same probability (see Figure 3.9(c)). As a result, some blocks are chosen and **swapped**, and others are **rotated** 90 degrees.



(a) Tournament selection

(b) Cycle crossover

(c) Swap mutation or rotation

Figure 3.9: *MOEA operators: Tournament selection, cycle crossover and two mutation operators (swap or rotate).*

3.3.2 Fitness function

Each chromosome represents the order in which blocks are being placed in the design area. Every block B_i is placed taking into account all the topological constraints, the total wire length, and the maximum temperature in the chip with respect to all the previously placed blocks $B_j : j < i$. In order to place a block i, we take the best point (x_i, y_i, z_i) in the remaining free positions. To select the best point we stablish a dominance relation taking into account the m following objectives in our multi-objective evaluation:

- The first objective is determined by the topological relations among placed blocks. It represents the number of topological constraints violated (no overlapping between placed blocks and current area less or equal than maximum area).
- The second objective is the wire length. The wire length is approximated as the Manhattan distance between interconnected blocks.
- The following objectives $(3, 4, \ldots, m)$ are a measures of the thermal impact, each one based on a power consumption besed on our profiles. In our case, we must obtain up to 600 different power consumptions (100 time windows $\times$ 6 applications).

Obviously, 600 objectives is too high for a MOEA, since it is almost impossible to converge. However, we discuss some ways to reduce this number of objectives in the next chapter. The compute the thermal impact for every power consumption we cannot use an accurate thermal model, which includes non-linear and differential equations. In a classical thermal model, the temperature of a unitary cell of the chip, depends not only on the power density dissipated by the cell, but also on the power density of its neighbors. The first factor refers to the increase of the thermal energy due to the activity of the element, while the second one is related to the diffusion process of heat [28]. Taking this into account, we use the power density of each block as an approximation of its temperature in the steady state. This is a valid approximation because the main term of the temperature of a cell is given by the power dissipated in the cell, the contribution of its neighbors does not change significantly the thermal behavior. Thus, our remaining objectives can be formulated as:

$$J_{k \in 3..m} = \sum_{i<j \in 1..n} (dp_i^{k-2} * dp_j^{k-2})/(d_{ij}) \tag{3.1}$$

where dp_i^p is the power density of block i for power consumption p, and d_{ij} is the Euclidean distance between blocks i and j.

In the next chapter, we explain how we obtain the power profiles for the different scenarios and benchmarks. Later on, in Chapter 5 we present the different parameters fixed to run this algorithm and explain the experiments proposed in this work.

Chapter 4

Power Profiling Phase

In the first section we give a description of the chosen benchmarks and the changes done to adapt them to be run by the OVPsim simulator. In the second section we explain how the energy profiling is carried out for the different memories and processors considered.

4.1 Benchmarks

This work approaches the thermal-guided floorplanning problem for manycore heterogeneous architectures. The temperature of a given chip depends on physical factors such as the power dissipation of the processors, the size of the memories etc. but it also depends on the dynamic profile of the applications. One of our contributions is to consider energy profiles based on the simulation of real world applications. In fact, this problem is generally approached considering only the worst case scenario in terms of power dissipation.

In this section we describe the benchmarks used and justify our choice. We work with ParMiBench [23] (Open-Source Benchmark for Multiprocessor Based Embedded Systems) which is composed of parallel versions of typical applications. We select 11 applications grouped into six different categories: Calculus, Network, Security, Office, Multimedia and Mixed. Therefore we have 6 different benchmarks corresponding to different kinds of applications that will exhibit very different execution profiles. As a result the power dissipation profiles obtained are different from one benchmark to another. A brief description of each of these benchmarks grouped into the different categories can be found below.

- **Calculus**: The applications forming this category are mainly mathematical intensive calculus applications such as solving cubic equations, converting values from decimal to radian, computing integer square roots, and bit counting in several different ways. In order to obtain representative energy profiles, we choose the four following applications:
  - a cubic equations solver

  - a deg to rad conversion
  - an integer square root
  - a bitcount application

- **Network**: The applications forming this category are typical algorithms used in graph analysis. In particular we choose a Dijkstra Shortest Path algorithm and a Patricia (Practical Algorithm to Retrieve Information Coded in Alphanumeric) algorithm working with IP addresses. In order to obtain representative energy profiles of network applications, we choose the two following applications:
  - Dijkstra Shortest Path algorithm
  - Patricia with IP addresses

- **Security**: The application forming this category implements a Secure Hash Algorithm (SHA), we consider only one benchmark for this category, enough to obtain a representative energy profile of security related applications.
  - SHA

- **Office**: The applications forming this category perform a string search in a given input text with two different methods. In order to obtain representative energy profiles of office applications, we choose the two following applications:
  - string search using the Boyer-Moore-Horspool method
  - string search using the Pratt-Boyer-Moore method

- **Multimedia**: The applications forming this category are multimedia applications working with images. These applications analyze the input image and produce a different image as output. In order to obtain representative energy profiles of multimedia applications, we choose the two following applications:
  - corner finder
  - image smoothing

- **Mixed**: We add an extra mixed profile that regroups two applications with a very different execution profile:
  - Calculus applications
  - Dijkstra shortest path algorithm

These applications are implemented with a shared memory strategy, in particular with Posix threads. This implementation is not the one we need to run the applications with OVPsim. It is due to the fact that we want to simulate these applications on bare machines (i.e. architectures that are not provided with an OS) to avoid the overhead caused by an

operating system in our power profiles. Therefore the benchmarks need to be adapted to be run by OVPsim (mapping the compiled code into specific memory regions, replacing the system calls etc.). The modification of these applications is intuitive: we replace the code defining the POSIX threads with functions of the OVPsim Api. Hence the code executed by a POSIX thread in the original ParMiBench will now be executed by a processor defined in the architecture.

As we do not have any automatic tool to manage the use of shared memory, we must assign manually a region to each of the executed programs. We also need to assign manually a specific memory address to every variable declared as shared. For some programs of the benchmark, the size of the shared memory region needed depends on the input data. Hence, we are forced to fix these inputs and discover the amount of memory needed with profiling techniques. It is important to note that if the inputs were not fixed, memory regions addressed by different processors could overlap. The accesses to the shared memory of the different processors are always performed in a sequential way due to the multiprocessor scheduling algorithm used. Hence, we do not need to care about the integrity of the data placed in shared memory because, during the simulations, only one processor at a time will access this memory. It is important to note that some of these programs only need to use shared flags while others need to share data, in some of them there is a lot of communication while in others the processors are practically independent. Therefore the access patterns to shared memory and the processor state will be very different from a given benchmark to another.

To obtain the profiling statistics for a given benchmark and a given architecture, we proceed as follows :

1. we fix the amount of processors that will execute a given application
2. we fix which processors will work together
3. we assign a shared memory region to each of the groups of processors working together

We illustrate this idea with an example, we show the task distribution of the Network benchmark for the three considered architectures. The distribution for the rest of the simulations is detailed in the Appendix A. With these task distributions, we aim to simulate scenarios that could happen on a real platform.

In the case of the 30 cores scenario (see Table 4.1), there are three groups of six processors (4 SPARC + 1 ARM + 1 PPC) computing the Dijkstra shortest path algorithm and 12 processors (8 SPARC + 2 ARM + 2 PPC) computing a Patricia algorithm.

In the 66 cores scenario (see Table 4.2), we can see 7 groups of six processors (2 SPARC + 2 ARM + 2 PPC) computing the Dijkstra shortest path algorithm and two groups of 12 processors (4 SPARC + 4 ARM + 4 PPC) computing a Patricia algorithm.

Finally, in the 129 cores scenario (see Table 4.3), there are fourteen groups of six processors (2 SPARC + 2 ARM + 2 PPC) computing the Dijkstra shortest path algorithm, three

Application	SPARC id	ARM id	PPC id
Patricia	0,1,2,3,4,5,6,7	20,21	25,26
Dijkstra1	8,9,10,11	22	27
Dijkstra2	12,13,14,15	23	28
Dijkstra3	16,17,18,19	24	29

Table 4.1: *Distribution of the Network benchmark for the 30 core architecture*

Application	SPARC id	ARM id	PPC id
Patricia1	0,1,2,3	22,23,24,25	44,45,46,47
Patricia2	4,5,6,7	26,27,28,29	48,49,50,51
Dijkstra1	8,9	30,31	52,53
Dijkstra2	10,11	32,33	54,55
Dijkstra3	12,13	34,35	56,57
Dijkstra4	14,15	36,37	58,59
Dijkstra5	16,17	38,39	60,61
Dijkstra6	18,19	40,41	62,63
Dijkstra7	20,21	42,43	64,65

Table 4.2: *Distribution of the Network benchmark for the 66 core architecture*

groups of 12 processors (4 SPARC + 4 ARM + 4 PPC) computing a Patricia algorithm and a group of 9 processors (3 SPARC + 3 ARM + 3 PPC) computing a Patricia algorithm.

4.2 Energy profiles

We use some of the features provided by the OVPsim software to obtain these profiles. As we showed in Chapter 2, the studied architectures are mainly composed of processors and memories: a local memory for each of the processors and a common shared memory. The objective of this section is to compute the power dissipated by each of these elements. We perform simulations of the chosen benchmarks in the different studied architectures. Therefore we need to obtain data from 3 * 6 = 18 simulations that provide us the statistical data we need to compute the power dissipated by each element. We study the evolution of the power dissipation over time for every element in the studied architecture. We proceed as follows:

1. We split the simulation of a given benchmark in time slices called windows
2. for each of these windows we obtain:
    - for each processor:
        - the amount of executed instructions
        - the amount of idle cycles

Application	SPARC id	ARM id	PPC id
Patricia1	0,1,2,3	43,44,45,46	86,87,88,89
Patricia2	4,5,6,7	47,48,49,50	90,91,92,93
Patricia3	8,9,10,11	51,52,53,54	94,95,96,97
Patricia4	12,13,14	55,56,57	98,99,100
Dijkstra1	15,16	58,59	101,102
Dijkstra2	17,18	60,61	103,104
Dijkstra3	19,20	62,63	105,106
Dijkstra4	21,22	64,65	107,108
Dijkstra5	23,24	66,67	109,110
Dijkstra6	25,26	68,69	111,112
Dijkstra7	27,28	70,71	113,114
Dijkstra8	29,30	72,73	115,116
Dijkstra9	31,33	74,75	117,118
Dijkstra10	33,34	76,77	119,120
Dijkstra11	35,36	78,79	121,122
Dijkstra12	37,38	80,81	123,124
Dijkstra13	39,40	82,83	125,126
Dijkstra14	41,43	84,85	127,128

Table 4.3: *Distribution of the Network benchmark for the 129 core architecture*

- for each memory (local and shared):
  - the amount of read accesses
  - the amount of write accesses

We obtain at least the statistical data of 100 windows. We fix the window period to $128ms$. This value is chosen to be long enough to reduce the impact on performance of the profiling phase, but short enough to capture the dynamic behavior with the required accuracy. The simulations are at least $12.8s$ long ($100 \times 0.128s$), enough to reach a stationary state.

4.2.1 Memories

We assume that the energy consumption of the memories depends on the amount of read/write accesses. The energy consumption of the memories depends on different factors such as memory size, line size, associativity, transistor size, page size etc. We obtain the energy consumption and area values with the CACTI software [20]. We approximate the energy per write access value with $E_w = E_r * 1.5$.

Local memories: The minimum size of the memories is found with profiling techniques. The application that needs the biggest amount of memory is the image smoothing where the studied image must be loaded into memory. If we limit the input images to small images we can run this application with $524288B \simeq 512KB$ of local memory (rounded up to the

nearest power of 2). As this applications is the worst case scenario, we fix the size of the local memories to $512KB$.

We consider the local memories to be direct mapped SRAM memories, with a block size of 64 bytes and a transistor size of 45 nm. We obtain the following values:

- Total dynamic read energy per access: 0.584846 nJ
- Total dynamic write energy per access: 1,364895 nJ
- Area: 2.65952 mm^2

Shared Memory: On the other hand, the size of the shared memory depends on the architecture we want to simulate. It is intuitive to think that a largest amount of processors will need a largest memory space to communicate and share data. Once again profiling techniques are used to find the minimum size of this memory in each of the three studied architectures. We fix the memory sizes as follows (rounded up to the nearest power of 2):

- $4194304B \simeq 4MB$ for the 30 cores architecture
- $8388608B \simeq 8MB$ for the 66 cores architecture
- $16777216B \simeq 16MB$ for the 129 cores architecture

We consider the shared memories to be a direct mapped SRAM memories, with a block size of 64 bytes and a transistor size of 45 nm. We obtain the following values:

- 30 cores architecture:
  - Total dynamic read energy per access: 2.37758 nJ
  - Total dynamic write energy per access: 3,56637 nJ
  - Area: 14.3553 mm^2
- 66 cores architecture:
  - Total dynamic read energy per access: 3.96696 nJ
  - Total dynamic write energy per access: 5,95044 nJ
  - Area: 28.256 mm^2
- 129 cores architecture:
  - Total dynamic read energy per access: 3.99656 nJ
  - Total dynamic write energy per access: 5,99484 nJ
  - Area: 57.513 mm^2

4.2.2 Processors

We assume that the energy consumption of a given processor depends on its working frequency and its state. We consider two states: active or idle. To compute the power densities of the processors, we need their areas and a power consumption value for both the active and idle states. We approximate the power dissipated in the idle state with $P_{idle} = P_{active}/10$.

- In [27] we find a power consumption of the SPARC of 4W at 1.4GHz and an area of $3.24mm^2$
- In the case of the CORTEX-A9[4], we find that 0.4W is the estimated power dissipation working at 830 MHz with an area of $1.5mm^2$
- The PPC440 9SF[22] dissipates 1.1W at 667MHz and has an area of $6.2mm^2$

4.2.3 Power profiles

Once we have the power consumption specifications of all the elements considered in this work, we need to obtain the dissipated power by each element in each time window.

In the case of the memories, it is immediate as we just need to multiply the different accesses counted during the simulations by the energy per access values retrieved with CACTI.

In the case of the processors, we proceed in a similar way. We need to compute the amount of active and idle cycles and multiply these values by the power consumption in the active and idle states respectively.

The floorplanner proposed in this work considers the power densities of the different components of our platforms. The power density of a given element i is computed with the following formula:

$$dp_i = \frac{p_i}{(length_i * cellsizeinμm * 10^{-6} * width_i * cellsizeinμm * 10^{-6})} \tag{4.1}$$

where p_i is the power consumption, $length_i$ and $width_i$ are the dimensions of the considered element and $cellsizeinμm$ is the size of the cell used by the floorplanner. The power density of each element and time window are used to guide the floorplanner as explained in the next chapter (see 5.1).

To summarize this section, we show in Table 4.4 the areas and power consumption values of all the elements considered in this work.

	area(mm^2)	l(mm)	w(mm)	en. r (nJ)	en. w (nJ)	p (W)	freq(GHz)
SPARC	3.24	1.800	1.800	-	-	4	1.4
Cortex-A9	1.5	1,225	1,225	-	-	0.4	0.830
PPC440	6.2	2,490	2,490	-	-	1.1	0.667
LMEM	2.660	1.129	2.357	0.584846	1,364895	-	-
SMEM30	14.355	3.208	4.475	2.37758	3,56637	-	-
SMEM66	28.256	3.155	8.957	3.96696	5,95044	-	-
SMEM129	57.513	6.404	8.981	3.99656	5,99484	-	-

Table 4.4: *Energy consumption and sizes of the different elements*

Chapter 5

Experimental Work

5.1 Experimental Setup

The experimental work will analyze the thermal optimizations achieved by the floorplanner in the three different proposed scenarios in 2.3.3 (30, 66 and 129 cores architectures). The floorplanner will place the processors, the local and shared memories of the 3D manycore platforms in 3, 4 and 5 layers respectively.

We compare four different floorplans obtained with our algorithm:

- As we do not have any original configuration to compare with, we propose as baseline a performance optimized floorplan targeting only the wire length. From now on, this configuration will be called BAS.

In order to obtain the thermally optimized floorplans, it is not possible to take directly into account all the data retrieved from our simulations. In fact, if the power dissipation of every element for every benchmark and time window was considered, the floorplanner would target 600 objectives and would hardly converge. Therefore, to obtain the other three floorplans, we consider different power metrics computed with the data retrieved from the simulation of 100 time windows for the 6 different execution profiles.

- The first of the remaining configurations is obtained considering the mean power dissipation for each element and profile (AVG). Therefore, the floorplanner looks for feasible solutions that minimize six thermal objectives (one per profile) and the wire length.
- Another configuration is obtained taking into account only the highest power consumption per element and profile (WOR). Hence, only two objectives are targeted, a thermal objective and the wire length. This case corresponds to the strategy followed by other thermal-aware floorplanners.
- Finally, a weighted sum of the power consumptions of the different profiles is considered for each element (WSM). In this case, the algorithm targets a thermal objective and the wire length.

We run the multi-objective genetic algorithm with a population of 100 individuals and 500 generations. These parameters are fixed according to previous research. The crossover probability p_c is fixed to 0.90 and the mutation probability p_m to $1/\#blocks$ (see [13]). We add 20% of extra area to the different layers to increase the efficacy of the algorithm.

The configurations obtained are chosen among a front of non-dominated solutions returned by the floorplanner. In the next section we explain how a solution of the non-dominated front is selected.

5.2 Results

As we explained in the previous section, the floorplanner returns a set of non-dominated solutions. The different solutions present in the returned front correspond to different floorplans that have different wire length and a different thermal behaviour. We first analyze the wire length of the different solutions obtained for the different configurations and scenarios. The wire length is the sum of the Manhattan distances between the connected elements. The baseline (BAS) corresponds to a configuration obtained targeting only the wire length. Therefore, it is an optimal configuration in terms of performance. We compare the wire length of the thermally optimized floorplans to the baseline, this way we can evaluate the tradeoff between thermal optimization and performance. Table 5.1 shows the maximum and minimum wire length obtained for the different solutions conforming a non-dominated front.

	BAS	*AVG*				*WOR*				*WSM*			
	d	d_{min}	%	d_{max}	%	d_{min}	%	d_{max}	%	d_{min}	%	d_{max}	%
30	1700	2327	36.88	2964	74.35	2095	23.24	2328	36.94	1923.5	13.15	2034	19.65
66	5643	6557	16.19	8639	53.09	7149	26.68	7949	40.87	6617	17.25	6794	20.40
129	15178	18943	24.81	20996	38.33	17863	17.69	20558	35.45	18564	22.31	19208	26.55

Table 5.1: *Wire overhead*

We can see that in the case of the AVG configuration, there is always a wide range of wire overhead. It is due to the fact that it is harder for the floorplanner to converge when it targets a high number of objectives (seven objectives). On the other hand, the solutions found for the WSM configuration are more homogeneous which is an indicator of the convergence of the algorithm. The WOR case corresponds to an intermediate result in terms of the convergence of the algorithm. **These first results show that a relaxation of the thermal constraints (WSM) leads to a faster convergence of the floorplanner.**

We also observe that some of the solutions obtained are not acceptable. For example, in the case of the AVG configuration for the 30 cores scenario, there is a wire overhead of 74.35%. Such a configuration implies a great loss of chip performance. To avoid such a performance penalty and to easily select a solution among the returned front of solutions, we choose in each case the configuration that presents the minimal wire overhead. Table 5.2 shows the wire overhead of the selected solutions. We can see that in each of the three

scenarios (30, 66 and 129 cores), the best wire overhead is found for a different configuration. In fact, the WSM configuration is the most efficient configuration for the 30 cores scenario, AVG outperforms the others in the 66 processors case while in the last scenario, the best performance is obtained by the WOR configuration.

	BAS		*AVG*		*WOR*		*WSM*	
	Dist	%	*Dist*	%	*Dist*	%	*Dist*	%
30	1700	0.00	2327	36.88	2095	23.24	1923.5	13.15
66	5643	0.00	6557	16.19	7149	26.68	6617	17.25
129	15178	0.00	18943	24.81	17863	17.69	18564	22.31

Table 5.2: *Wire Overhead*

As further analysis will show, solutions with a reduced wire length tend to have a poor thermal behaviour. On the other hand, the higher the wire length, the better the thermal response of a given solution. In this section, we compare the thermal profiles of the different configurations obtained in 5.1 and propose a way to deal with the performance/temperature tradeoff. We propose two experiments, in the first one we evaluate the thermal response of the different configurations in the worst case scenario. In the second experiment, the thermal behaviour of these configurations is studied with the real power profiles retrieved from our simulations.

5.2.1 Worst Case Scenario

In the first experiment, the thermal simulator evaluates the different floorplans with the highest power dissipation for every element. This case corresponds to the worst scenario. The metrics considered for the analysis of the experimental results are the mean and maximum temperature of the chip and the maximum thermal gradient. These metrics are usually found in thermal-related analysis. In table 5.3, 5.4 and 5.5 we present the thermal profiles of the four different configurations.

	$WireOverhead$	T_{max}	T_{mean}	$Grad_{Max}$
BAS	0%	416.30	355.22	108.44
AVG	36.88%	394.80	350.10	84.93
WOR	23.24%	388.42	349.03	73.46
WSM	13.15%	387.68	349.73	74.12

Table 5.3: *Worst case 30 cores*

The results show that our power profiling-guided floorplanner produces thermally optimized configurations. **The hotspots found in the performance optimized floorplans justify the thermal optimization presented in this work.** Compared to the baseline, we can see that in all the cases our floorplanner reduces the peak and mean temperatures and the thermal gradient. Therefore, not only the temperature of the chip is reduced but it

	$WireOverhead$	T_{max}	T_{mean}	$Grad_{Max}$
BAS	0%	440.24	357.35	136.10
AVG	16.19%	410.187	352.41	101.92
WOR	26.68%	401.27	349.55	91.22
WSM	17.25%	403.34	349.40	91.742

Table 5.4: *Worst case 66 cores*

	$WireOverhead$	T_{max}	T_{mean}	$Grad_{Max}$
BAS	0%	443.15	361.56	137.57
AVG	24.81%	396.33	355.32	87.06
WOR	17.69%	414.45	355.98	104.59
WSM	22.31%	400.44	354.63	91.75

Table 5.5: *Worst case 129 cores*

is more evenly distributed. For example, we can appreciate that the dramatic peak temperature of 416.30K found in the baseline is reduced to 387.68K in the WSM configuration in the 30 core platform. We illustrate this example in Figure 5.1 where we show the thermal maps of the first layer of the BAS and WSM configurations. In the baseline configuration the floorplanner tends to place the processors near their local memories ignoring the presence of hotspots. In the other the hottest elements (the SPARC cores) are separated as much as possible, generally placed in the borders of the chip. As a consequence, we observe a much better thermal response of the WSM configuration. Vertical heat spread is also taken into account, hence the floorplanner avoids placing cores above the others. In both cases the shared memory is placed in the second layer to minimize the wire length.

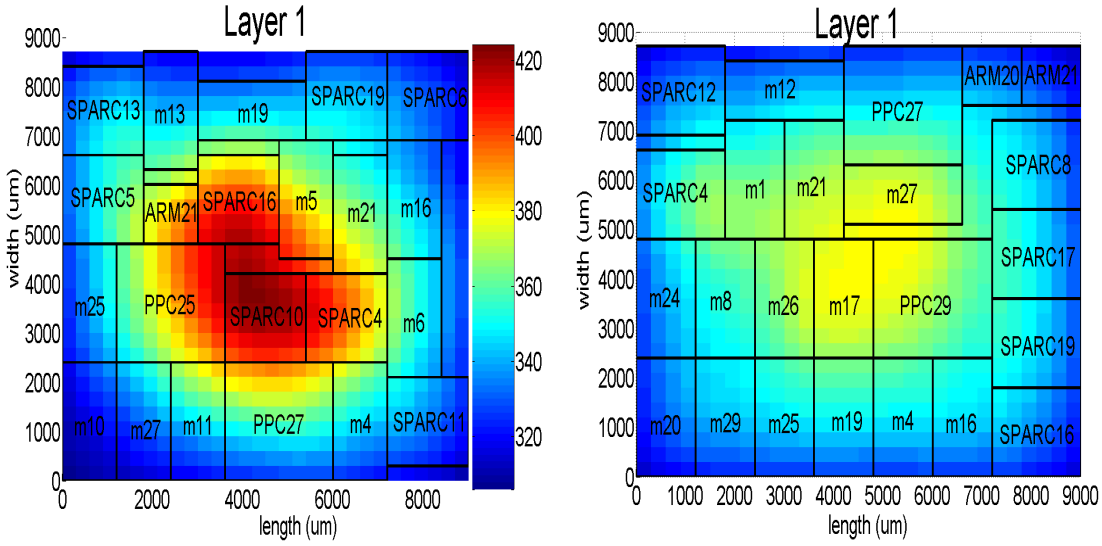


Figure 5.1: *Thermal map of the first layer of the 30 cores BAS(left) and WSM(right) configurations*

The baseline configuration (BAS) is not an acceptable solution, in fact it reaches peaks of 443.15K in the 129 cores configuration. Moreover, the peak temperature increases with the number of cores in all of the cases. Therefore, from now on we will use the baseline to compare the wire length of the thermally optimized floorplans, but we will not study its thermal response.

Choosing a configuration between the thermally optimized floorplans is not immediate. It is due to the fact that some of these configurations present a better performance (wire length) while others have a better thermal behaviour. We propose a selection criterion in 5.2.3.

5.2.2 Real power Profiles

In the second experiment proposed, we evaluate the floorplans in a more realistic way, the thermal behaviour of the different configurations is simulated for 100 time windows with the power dissipation values obtained from our execution profiles. Three metrics are considered in this case to compare the different configurations:

- The mean of the maximum temperatures of the different time windows is given as well as its standard deviation. This metric is a good indicator of the existence of hotspots in the studied chip.
- We also compute the overall mean temperature of the chip and its standard deviation.
- The mean of the maximum thermal gradients and its standard deviation is presented as well.

In Table 5.6 (30 cores scenario), Table 5.7 (66 cores scenario) and Table 5.8 (129 cores scenario) we present these metrics for the six studied benchmarks and the three different configurations.

Table 5.6: *30 cores architecture*

	Wire	Tmax					
		BENCH1	BENCH2	BENCH3	BENCH4	BENCH5	BENCH6
AVG	36.88%	$340.95_{6.58}$	$342.55_{4.56}$	$340.22_{7.80}$	$310.35_{3.15}$	$348.33_{2.68}$	$343.75_{3.38}$
WOR	23.24%	$338.73_{7.44}$	$345.51_{2.66}$	$341.12_{11.09}$	$311.43_{2.82}$	$347.48_{2.60}$	$340.29_{3.22}$
WSM	13.15%	$335.93_{6.30}$	$345.56_{3.79}$	$340.80_{10.02}$	$309.37_{2.89}$	$346.65_{2.56}$	$341.38_{2.63}$
	Wire	Tmean					
		BENCH1	BENCH2	BENCH3	BENCH4	BENCH5	BENCH6
AVG	36.88%	$322.43_{6.14}$	$320.11_{2.26}$	$324.70_{5.59}$	$306.28_{1.95}$	$331.41_{1.65}$	$325.56_{2.86}$
WOR	23.24%	$322.19_{5.79}$	$319.82_{2.19}$	$324.33_{5.49}$	$306.24_{1.91}$	$330.29_{1.58}$	$324.80_{2.70}$
WSM	13.15%	$322.50_{6.07}$	$319.96_{2.36}$	$324.64_{5.65}$	$306.21_{1.98}$	$331.37_{1.64}$	$325.49_{2.95}$
	Wire	Tgrad					
		BENCH1	BENCH2	BENCH3	BENCH4	BENCH5	BENCH6
AVG	36.88%	$35.88_{5.83}$	$39.45_{4.58}$	$34.40_{6.32}$	$8.47_{2.63}$	$42.70_{2.43}$	$38.73_{3.60}$
WOR	23.24%	$31.85_{6.25}$	$40.66_{2.65}$	$32.25_{7.97}$	$9.54_{1.95}$	$36.52_{2.08}$	$31.57_{4.45}$
WSM	13.15%	$29.36_{4.44}$	$41.99_{3.99}$	$34.05_{8.39}$	$7.03_{2.24}$	$36.59_{2.10}$	$34.18_{2.83}$

30 cores scenario: The results obtained for the peak temperatures are not conclusive as none of the configurations clearly outperforms the others. The highest difference is obtained in the case of the first Benchmark where the temperature reached by the WSM is 2.8°C and 5.02°C lower than the one obtained with the WOR and AVG configurations respectively. The mean temperatures are very similar for all the configurations. In fact there is a maximum difference of 1.12°C between the different configurations (Benchmark 5). As for the peak temperatures, a first analysis of the maximum thermal gradients does not lead to an immediate selection of the best configuration. Further analysis of these results is proposed in 5.2.3 where we find the configuration that minimizes wire length and thermal metrics for each benchmark.

Table 5.7: *66 cores architecture*

	Wire	Tmax					
		BENCH1	BENCH2	BENCH3	BENCH4	BENCH5	BENCH6
AVG	16.19%	$334.17_{9.35}$	$365.43_{7.36}$	$386.49_{3.14}$	$369.64_{1.64}$	$349.52_{3.12}$	$369.16_{4.64}$
WOR	26.68%	$329.77_{5.60}$	$366.91_{6.16}$	$385.90_{5.75}$	$369.60_{1.94}$	$342.91_{2.63}$	$368.72_{5.68}$
WSM	17.25%	$331.40_{6.28}$	$364.66_{7.52}$	$383.27_{1.96}$	$369.33_{1.62}$	$342.30_{2.62}$	$367.96_{5.62}$
	Wire	Tmean					
		BENCH1	BENCH2	BENCH3	BENCH4	BENCH5	BENCH6
AVG	16.19%	$319.61_{5.22}$	$325.63_{1.06}$	$338.79_{3.43}$	$324.90_{0.60}$	$328.27_{1.51}$	$324.63_{0.62}$
WOR	26.68%	$318.36_{4.16}$	$325.91_{0.92}$	$337.96_{3.81}$	$324.77_{0.54}$	$325.23_{1.36}$	$324.51_{0.71}$
WSM	17.25%	$318.57_{4.40}$	$325.27_{0.96}$	$337.29_{3.18}$	$324.33_{0.55}$	$325.40_{1.36}$	$323.83_{0.65}$
	Wire	Tgrad					
		BENCH1	BENCH2	BENCH3	BENCH4	BENCH5	BENCH6
AVG	16.19%	$29.50_{8.42}$	$62.35_{7.65}$	$80.75_{2.22}$	$66.99_{1.87}$	$44.48_{3.04}$	$66.25_{4.98}$
WOR	26.68%	$24.75_{3.95}$	$64.35_{6.60}$	$79.99_{5.51}$	$67.35_{2.24}$	$35.35_{2.56}$	$66.42_{6.43}$
WSM	17.25%	$26.02_{4.93}$	$61.41_{7.94}$	$76.90_{1.23}$	$66.91_{2.02}$	$34.43_{2.38}$	$65.45_{6.59}$

66 cores scenario: We can see that the peak temperatures obtained in the WSM are lower than those obtained in the AVG configuration saving up to 7.22°C. Only in the case of the Benchmark 1 the peak temperature of the WOR configuration is the lowest one, with a difference of 1.63°C over the WSM. On the other hand, selecting WSM leads to a maximum reduction of 2.63°C in the peak temperature for the Benchmark 3. Once again, there are no significant differences between the mean temperatures obtained for the studied configurations. In the case of the maximum thermal gradients, the results are similar to those obtained for the peak temperature: the WSM outperforms the other configurations in five out of six cases, reducing up to 10.05°C the gradient of the AVG (Benchmark 5) and 3.09°C the gradient of the WOR (Benchmark 3). In the case of the Benchmark 1, the best thermal gradient is obtained for the WOR configuration, with a difference of 1.27°C and 4.75°C with the WSM and AVG configurations respectively.

129 cores scenario: The best peak temperature is obtained for the AVG configuration in four out of six benchmarks saving up to 17.12°C and 6.71°C comparing with the WOR (Bench 5) and WSM (Bench 1) configurations respectively. The WSM outperforms the others in the remaining two cases, reducing up to 10.80°C the maximum temperature of the WOR configuration and 2.98°C the one of the AVG. The mean temperatures obtained

Table 5.8: *129 cores architecture*

	Wire	Tmax					
		BENCH1	BENCH2	BENCH3	BENCH4	BENCH5	BENCH6
AVG	24.81%	$342.31_{7.94}$	$357.95_{0.78}$	$373.18_{4.97}$	$352.95_{1.50}$	$370.42_{4.13}$	$331.63_{2.46}$
WOR	17.69%	$354.08_{8.90}$	$363.76_{5.07}$	$382.00_{9.27}$	$352.40_{2.51}$	$387.54_{5.19}$	$338.28_{2.93}$
WSM	22.31%	$349.02_{4.73}$	$363.99_{0.87}$	$371.20_{4.10}$	$351.14_{1.54}$	$373.87_{4.34}$	$336.10_{2.68}$
	Wire	Tmean					
		BENCH1	BENCH2	BENCH3	BENCH4	BENCH5	BENCH6
AVG	24.81%	$327.32_{5.80}$	$328.68_{1.34}$	$338.78_{4.53}$	$321.70_{1.86}$	$344.24_{2.36}$	$322.20_{1.57}$
WOR	17.69%	$328.25_{5.95}$	$329.19_{1.49}$	$339.01_{4.77}$	$321.57_{1.89}$	$344.87_{2.40}$	$322.57_{1.61}$
WSM	22.31%	$327.99_{5.52}$	$328.52_{1.43}$	$338.20_{4.99}$	$320.56_{1.98}$	$344.12_{2.35}$	$322.55_{1.59}$
	Wire	Tgrad					
		BENCH1	BENCH2	BENCH3	BENCH4	BENCH5	BENCH6
AVG	24.81%	$36.08_{6.31}$	$54.77_{1.00}$	$66.81_{3.52}$	$51.02_{1.07}$	$62.04_{3.77}$	$25.86_{2.14}$
WOR	17.69%	$47.75_{7.79}$	$60.06_{4.79}$	$75.95_{8.28}$	$50.70_{1.97}$	$78.82_{4.80}$	$32.12_{2.56}$
WSM	22.31%	$43.71_{3.80}$	$61.04_{0.91}$	$65.84_{3.22}$	$49.45_{1.14}$	$66.28_{3.98}$	$31.90_{2.49}$

are again homogeneous. The results for the the thermal gradients are similar to the ones obtained for the peak temperature. Globally, the WOR configuration shows a worst thermal behavior than the WSM or AVG. Nonetheless, it is the configuration that presents the best performance (wire overhead), therefore it can not be discarded.

Further analysis is required in all the scenarios studied to obtain the optimum configuration. In the next section we propose a method to select the optimum solution for the three scenarios taking into account the wire length and the thermal behavior of the different configurations.

5.2.3 Performace/temperature tradeoff

As seen in the previous subsection, there is a tradeoff between performance and chip temperature. We propose a method to evaluate the different solutions and select the one with the best overall behaviour. For each of the scenarios and metrics studied ($wire, T_{Max}, T_{mean}$ and $Grad_{Max}$) we establish a confidence interval. In a second step we see wether or not the retrieved metrics fall into these ranges of acceptable values.

The confidence intervals are obtained by adding and subtracting a given percentage to the mean of the metric considered. This percentage is different for each metric, as the ranges of the values obtained are very different from a metric to another. We fix a percentage of:

- T_{Max} and T_{Mean} metrics: we fix a percentage of 1%, this parameter results in confidence intervals of approximately 6°C
- $Grad_{Max}$ metric: we fix a percentage of 5% also resulting in confidence intervals of approximately 6°C
- $Wire$ overhead metric: we fix a percentage of 10% resulting in confidence intervals of approximately 4 units (percentage relative to the wire overhead comparing with the baseline configuration)

As an example, we compute these intervals for the worst case scenario:

- 30 cores architecture
  - Wire overhead: we obtain the mean of the wire overhead of the different configurations, which is 24.42%. The confidence interval is obtained by adding and subtracting a 10%, the resultant interval is :
    $mean = 24.42\% \rightarrow [24.42 - 10\%; 24.42 + 10\%] \simeq [21.98; 26.86]$
  - T_{Max}: in a similar way, we obtain the range of acceptable peak temperatures by adding and subtracting 1% of the mean of the maximum temperatures of the different configurations:
    $mean = 390.2 \rightarrow [390.2 - 1\%; 390.2 + 1\%] \simeq [386.30; 394.10]$
  - T_{mean}: The same method is used to obtain the confidence interval for the mean temperatures:
    $mean = 349.62 \rightarrow [349.62 - 1\%; 349.62 + 1\%] \simeq [346.12; 353.12]$
  - $Grad_{Max}$: the range of acceptable values for the maximum thermal gradient is obtained by adding and subtracting 5% to the mean of the maximum thermal gradients($77.5K$) of the different floorplans.
    $mean = 77.5 \rightarrow [77.5 - 5\%; 77.5 + 5\%] \simeq [73.63; 81.375]$

The same method is used for the 66 and 129 cores scenarios. We show the mean of the considered metric and the corresponding confidence interval:

- 66 cores architecture:
  - Wire overhead: $mean = 20.04\% \rightarrow [18.04; 22.04]$
  - T_{Max}: $mean = 404.93 \rightarrow [404.93 - 1\%; 404.93 + 1\%]$
  - T_{mean}: $mean = 350.45 \rightarrow [350.45 - 1\%; 350.45 + 1\%]$
  - $Grad_{Max}$: $mean = 94.96 \rightarrow [94.96 - 5\%; 94.96 + 5\%]$
- 129 cores architecture
  - Wire overhead: $mean = 21.60\% \rightarrow [19.44; 23.76]$
  - T_{Max}: $mean = 403.74 \rightarrow [403.74 - 1\%; 403.74 + 1\%]$
  - T_{mean}: $mean = 355.31 \rightarrow [355.31 - 1\%; 355.31 + 1\%]$
  - $Grad_{Max}$: $mean = 94.47 \rightarrow [94.47 - 5\%; 94.47 + 5\%]$

These intervals allow us to put into the same level similar values, in fact a difference of 1°C might not relevant to decide which configuration is better. On the other hand, stepping from a 15% of area overhead to 25% would result in a dramatic decrease of the chip performance.

Once we have these intervals, we see which metrics fall into these intervals (marked as $\surd$ in the following tables) and which ones do not (X). There is a third possibility where the metric considered is even below the confidence interval(marked as $\surd\surd$), which is the

desirable situation as the goal is to minimize the wire length and the different thermal metrics. The following tables show the result of this analysis for the worst case scenario and the 6 benchmarks. We perform an analysis based on the obtained confidence intervals and decide which is the best configuration for each of the considered cases.

Worst Case

Table 5.9 shows the results of the analysis based on the confidence intervals in the worst case scenario. This case corresponds to the highest power dissipation possible for the studied chips. Hence it is representative of the chip response in extreme conditions.

30	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	X	$\surd$	X
WOR	$\surd$	$\surd$	$\surd$	$\surd\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
66	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	$\surd\surd$	X	$\surd$	X
WOR	X	$\surd$	$\surd$	$\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
129	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd\surd$	$\surd$	$\surd\surd$
WOR	$\surd\surd$	X	$\surd$	X
WSM	$\surd$	$\surd$	$\surd$	$\surd$

Table 5.9: *Worst case*

30 cores scenario: We can see that only the WOR and the WSM configurations satisfy all the constraints. The WSM has 10.09% less of of wire overhead and thus, presents a greater performance. On the other hand, the WOR configuration presents a better thermal gradient, reducing in 2.61°C the gradient of the WSM. Therefore, both the WSM and WOR configurations are chosen in this case.

66 cores scenario: In this case, the WSM is the only configuration with acceptable values for all the metrics. Hence it is the selected configuration.

129 cores: In this case, the WSM is also the only configuration that satisfies all the constraints, therefore we select the WSM.

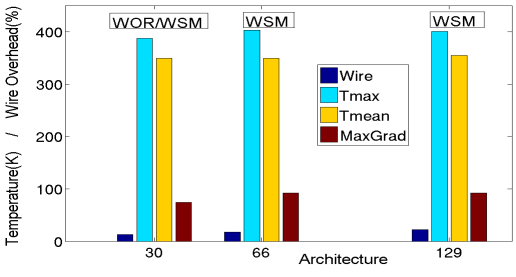


Figure 5.2: *Selected configurations in the Worst Case*

Benchmark 1

Table 5.10 shows the results of the analysis based on the confidence intervals obtained for the Benchmark 1. This case corresponds to the power dissipation of the studied chips while executing the MATH benchmark.

30	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd$	$\surd$	X
WOR	$\surd$	$\surd$	$\surd$	$\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd\surd$
66	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	$\surd\surd$	$\surd$	$\surd$	X
WOR	X	$\surd$	$\surd$	$\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
129	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd\surd$	$\surd$	$\surd\surd$
WOR	$\surd\surd$	X	$\surd$	X
WSM	$\surd$	$\surd$	$\surd$	$\surd$

Table 5.10: *BENCHMARK1*

30 cores scenario: We can see that the WOR and the WSM configurations correspond to acceptable solutions. Nevertheless WSM presents a better performance and thermal gradient. Therefore, WSM is the best configuration.

66 cores scenario: In this case, the WSM is the only configuration with accepatable values for all the metrics. Hence it is the selected configuration.

129 cores: The WSM is also the only configuration that satisfies all the constraints, therefore we select the WSM.

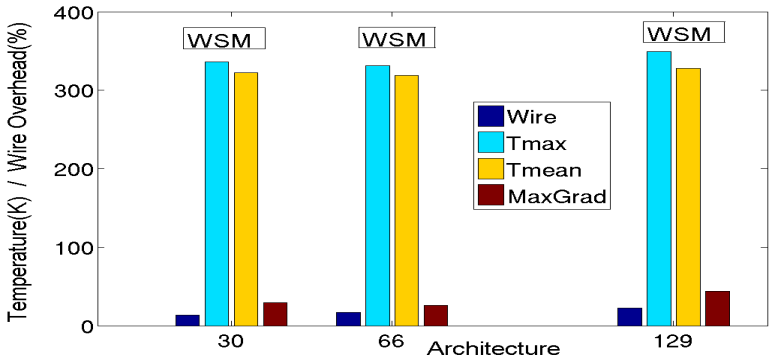


Figure 5.3: *Selected configurations for the Bench 1*

Benchmark 2

Table 5.11 shows the results of the analysis based on the confidence intervals obtained for the Benchmark 2. This case corresponds to the power dissipation of the studied chips while executing the MIXED benchmark.

30	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd$	$\surd$	$\surd$
WOR	$\surd$	$\surd$	$\surd$	$\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
66	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	$\surd\surd$	$\surd$	$\surd$	$\surd$
WOR	X	$\surd$	$\surd$	$\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
129	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd\surd$	$\surd$	$\surd\surd$
WOR	$\surd\surd$	$\surd$	$\surd$	$\surd$
WSM	$\surd$	$\surd$	$\surd$	$\surd$

Table 5.11: *BENCHMARK2*

30 cores scenario: We can see that the WOR and the WSM configurations correspond to acceptable solutions. WSM presents a better performance. Hence, WSM is selected as the best configuration.

66 cores scenario: In this case, both the AVG and WSM satisfy all the constraints equally. Hence they are both selected.

129 cores: The WOR and WSM satisfy all the constraints. Nevertheless the performance offered by the WOR configuration outperforms the WSM. Therefore we select the WOR configuration.

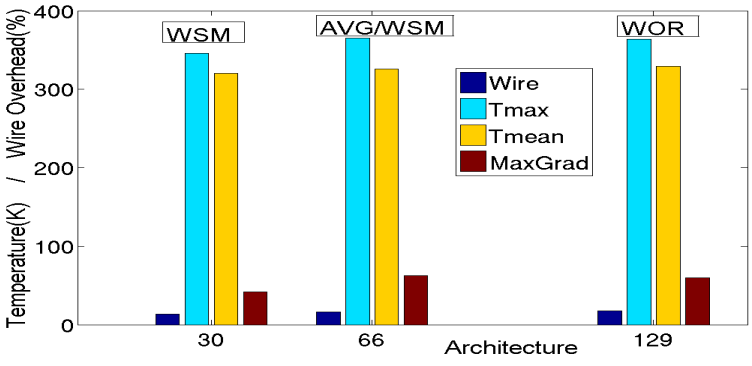


Figure 5.4: *Selected configurations for the Bench 2*

Benchmark 3

Table 5.12 shows the results of the analysis based on the confidence intervals obtained for the Benchmark 3. This case corresponds to the power dissipation of the studied chips while executing the MULTIMEDIA benchmark.

30	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd$	$\surd$	$\surd$
WOR	$\surd$	$\surd$	$\surd$	$\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
66	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	$\surd\surd$	$\surd$	$\surd$	$\surd$
WOR	X	$\surd$	$\surd$	$\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
129	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd$	$\surd$	$\surd$
WOR	$\surd\surd$	X	$\surd$	X
WSM	$\surd$	$\surd\surd$	$\surd$	$\surd$

Table 5.12: *BENCHMARK3*

30 cores scenario: We can see that the WOR and WSM configurations are acceptable solutions. Hence, WSM is selected as the best configuration as it presents a better performance.

66 cores scenario: In this case, AVG and WSM satisfy all the constraints equally. Therefore we select them both.

129 cores: The WSM is the only configuration to satisfy all the constraints, therefore we select the WSM.

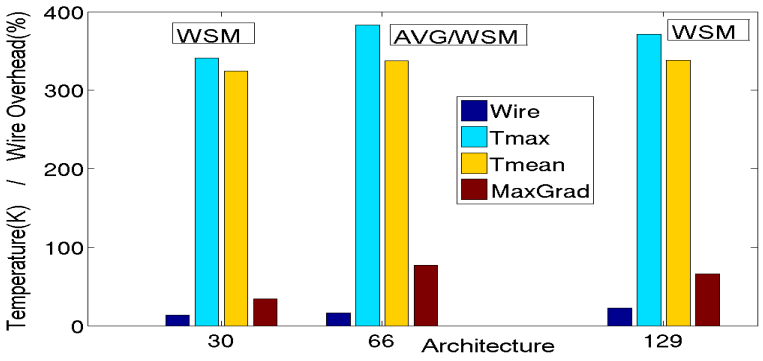


Figure 5.5: *Selected configurations for the Bench 3*

Benchmark 4

Table 5.13 shows the results of the analysis based on the confidence intervals obtained for the Benchmark 4. This case corresponds to the power dissipation of the studied chips while executing the NETWORK benchmark.

30	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd$	$\surd$	$\surd$
WOR	$\surd$	$\surd$	$\surd$	X
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd\surd$
66	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	$\surd\surd$	$\surd$	$\surd$	$\surd$
WOR	X	$\surd$	$\surd$	$\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
129	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd$	$\surd$	$\surd$
WOR	$\surd\surd$	$\surd$	$\surd$	$\surd$
WSM	$\surd$	$\surd$	$\surd$	$\surd$

Table 5.13: *BENCHMARK4*

30 cores scenario: The WSM is the only configuration that satisfies all the constraints, therefore it is selected.

66 cores scenario: AVG and WSM satisfy all the constraints equally. Therefore we select both of them as best configurations.

129 cores: We can see that WOR and WSM configurations correspond to acceptable solutions. WOR is selected as it presents a better performance.

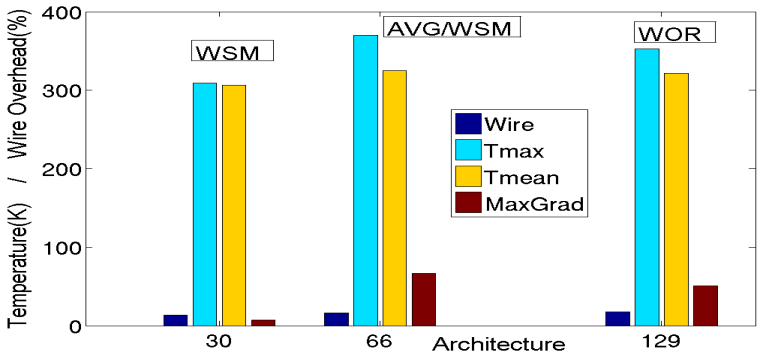


Figure 5.6: *Selected configurations for the Bench 4*

Benchmark 5

Table 5.14 shows the results of the analysis based on the confidence intervals obtained for the Benchmark 5. This case corresponds to the power dissipation of the studied chips while executing the OFFICE benchmark.

30	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd$	$\surd$	X
WOR	$\surd$	$\surd$	$\surd$	$\surd\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd\surd$
66	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	$\surd\surd$	X	$\surd$	X
WOR	X	$\surd$	$\surd$	$\surd\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd\surd$
129	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd\surd$	$\surd$	$\surd\surd$
WOR	$\surd\surd$	X	$\surd$	X
WSM	$\surd$	$\surd$	$\surd$	$\surd$

Table 5.14: *BENCHMARK5*

30 cores scenario: We can see that WOR and WSM configurations correspond to acceptable solutions. WSM is selected as it presents a better performance.

66 cores scenario: The WSM is the only configuration that satisfies all the constraints. Hence, it is selected.

129 cores: The WSM is the only acceptable configuration. Therefore we select WSM as the best configuration.

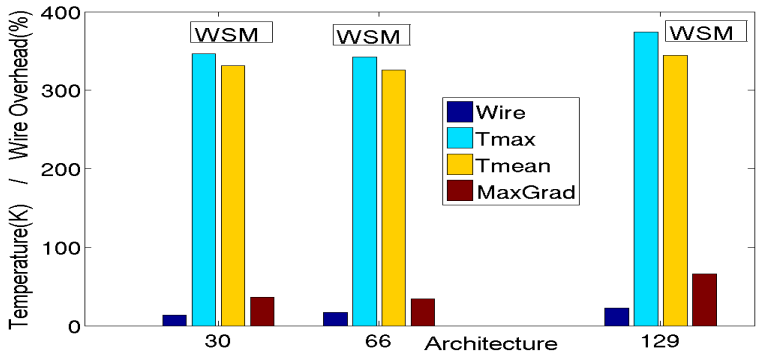


Figure 5.7: *Selected configurations for the Bench 5*

Benchmark 6

Table 5.15 shows the results of the analysis based on the confidence intervals obtained for the Benchmark 6. This case corresponds to the power dissipation of the studied chips while executing the SECURITY benchmark.

30	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd$	$\surd$	X
WOR	$\surd$	$\surd$	$\surd$	$\surd\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
66	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	$\surd\surd$	$\surd$	$\surd$	$\surd$
WOR	X	$\surd$	$\surd$	$\surd$
WSM	$\surd\surd$	$\surd$	$\surd$	$\surd$
129	$Wire$	T_{max}	T_{mean}	$Grad_{Max}$
AVG	X	$\surd\surd$	$\surd$	$\surd\surd$
WOR	$\surd\surd$	$\surd$	$\surd$	X
WSM	$\surd$	$\surd$	$\surd$	X

Table 5.15: *BENCHMARK6*

30 cores scenario: We can see that only the WOR and the WSM configurations satisfy all the constraints. The WSM has presents a better performance. On the other hand, the WOR configuration presents a better thermal gradient. Therefore, both the WSM and WOR configurations are chosen in this case.

66 cores scenario: The AVG and WSM configurations satisfy all the constraints equally. Therefore we select both of them.

129 cores: In this case, none of the configurations satisfies all the constraints. Nevertheless the WOR and WSM do not satisfy the thermal gradient constraint, i.e. they fall out of the interval by 0.66°C and 0.44°C respectively which is not significant. Moreover, their peak temperatures reached in this benchmark are low: 338.28K for the WOR and 336.10K for the WSM configurations. Therefore, we choose a solution between the WOR and WSM solutions. WOR presents a better performance, therefore it is selected.

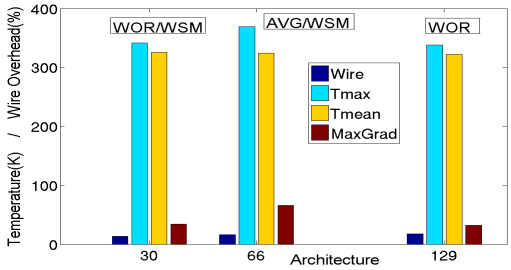


Figure 5.8: *Selected configurations for the Bench 6*

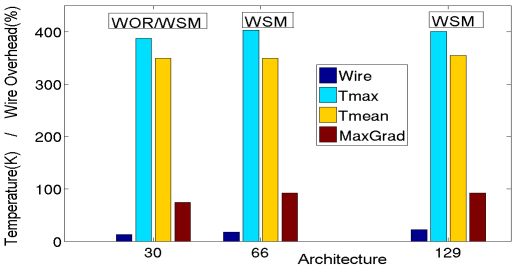
5.2.4 Results summary

In this section we summarize the results of the experiments proposed in the previous sections. Figure 5.9 shows the configurations chosen for the different benchmarks and scenarios. We analyze the configuration with the best overall behavior in each scenario.

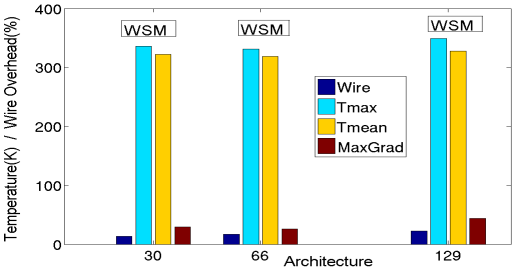
30 cores scenario: In this scenario only the WSM configuration is chosen for all the benchmarks. It offers a satisfactory performance while respecting the different thermal metrics studied in this work. The WOR configuration is also selected in two out of 7 cases (Worst Case and Bench 6). Moreover, this configurations respects the different constraints in all the studied cases. However the WSM offers a better performance as it presents 10.09% less of wire overhead. Therefore **the WSM is the best configuration for the 30 cores scenario**.

66 cores scenario: The WSM configuration is also chosen in all the cases. It presents the best overall thermal behavior while only the AVG presents a slightly better performance (1.06%). The AVG configuration presents the best performance and is also chosen in four out of seven cases but it does not satisfy the thermal constraints in the Worst and Benchmark 5 cases. Therefore it offers a poor response in extreme situations, leading to hotspots. Hence **the WSM is also chosen in this scenario as the best configuration**.

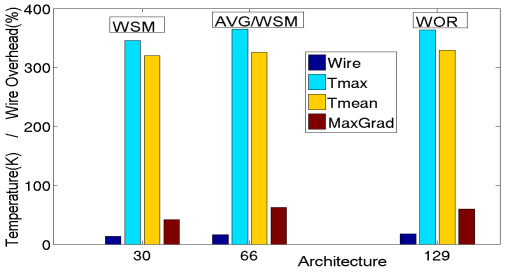
129 cores: The selection of the best configuration is not immediate as in the previous scenarios. In fact, the WOR configuration is chosen in three out of seven cases (Benchmarks 2, 4 and 6) that correspond to the MIXED, NETWORK and SECURITY benchmarks. These data-oriented benchmarks are not loop-dominated, in fact they are executed in a sequential manner presenting an elevated number of branch instructions. In addition, power consumption is distributed in an homogeneous way during the execution of these kind of applications which makes the Worst Case approximate well the dissipated power in these cases. Nevertheless this configuration violates the peak temperature and thermal gradient constraints in four of the seven studied cases (Worst, Benchmarks 1,3,5). On the other hand the WSM configuration is chosen in four out of seven cases and presents acceptable solutions in all the cases except for Benchmark 6, where none of the configurations respects all the constraints. In that particular case, the analysis shows that it fails to minimize the thermal gradient by 0.44°C and the peak temperature reached is not critical ($331.6K \simeq 63.1^{o}C$), therefore the solution remains acceptable. Hence, the WSM is the configuration that presents best overall behavior. The **WSM configuration is also chosen in this scenario**.



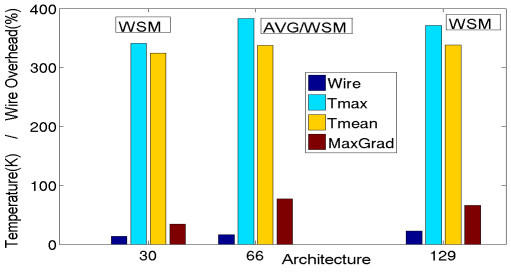
(a) WORST CASE



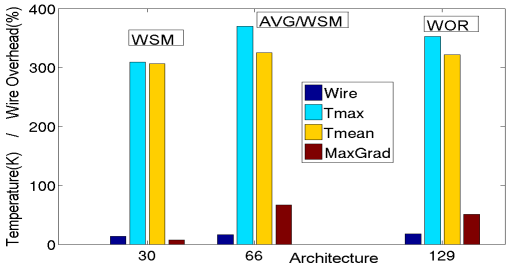
(b) Benchmark 1



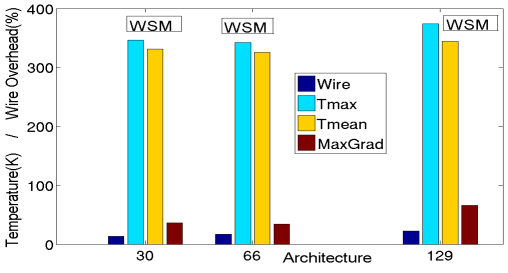
(c) Benchmark 2



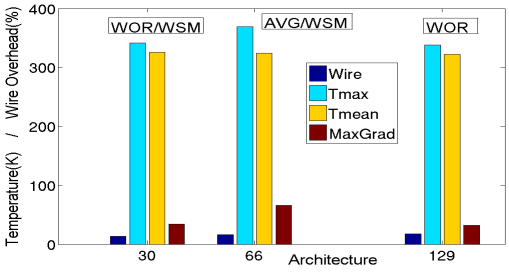
(d) Benchmark 3



(e) Benchmark 4



(f) Benchmark 5



(g) Benchmark 6

Figure 5.9: *Best Configuration for each Benchmark*

Chapter 6

Conclusions and Future Work

In this work we propose for a first time a scalable efficient thermal-aware 3D floorplanner for heterogeneous architectures of MPSoCs that uses as input the power traces obtained during an application power profiling phase.

The power profiles are retrieved from simulations of manycore heterogeneous architectures and correspond to the dynamic profiles of different real world applications. These simulations are carried out with the multiprocessor platform emulator OVPsim.

We design three manycore heterogeneous architectures inspired in Intel's Single-Chip Cloud Computer. The designed heterogeneous platforms are composed of SPARC cores and the low power oriented processors ARM Cortex-A9 and PPC440 9SF. Our architectures are composed of 30, 66 and 129 cores with their local memories and a common shared memory used for inter-processor communication. Such manycore heterogeneous architectures are expected to be the future trend in computer design.

The ParMiBench suite is adapted to be run on bare machines and simulated with OVPsim to avoid the power overhead caused by an operating system. This suite contains several Calculus, Network, Security, Office and Multimedia applications that exhibit very different execution profiles.

We propose a scalable Multi-Objective Genetic Algorithm based on the NSGA-II capable of proposing floorplans for our architectures composed of 30, 66 and 129 cores. Previous research shows that MILP based techniques are unable to propose solutions in the latter case. We propose a representation of the solutions and genetic operators that lead to optimal floorplans, in a short time, for a large number of integrated processors and layers and with minimal overhead.

The proposed multi-objective genetic algorithm returns different floorplans by targeting up to six different thermal objectives and the wire length. We evaluate these configurations with real power values retrieved from the simulation of 6 different benchmarks.

The proposed analysis of the tradeoff between thermal behavior and performance shows that considering the worst power consumption does not lead to optimal floorplans. In fact, when we take into account performance and thermal metrics, a floorplan that tries to minimize the weighted sum of the power consumption of the different benchmarks shows a better overall behavior. Such configuration presents a better thermal response in extreme conditions and reduces in 13,67°C and 12,54°C the peak temperature and the thermal gradient of the chip respectively. Therefore, the effect of hotspots is reduced and the temperature of the chip is more evenly distributed.

We show that considering the worst case in terms of power consumption leads to non optimal floorplans due to an overestimation of the temperature of the chip. Therefore, a power profiling phase of representative applications that will run in the considered architectures is necessary to find thermally optimized solutions.

We have found a metric (WSM) that is able to outperform the results of the traditionally used WOR (Worst case) for a set of representative benchmarks of the application scope.

The choice of the optimum floorplan among the set of non-dominated solutions returned by the floorplanner is postponed as a future work.

Future research also expects to find a lower number of meaningful windows to reduce the overhead of profiling.

Finally, a better representation of the thermal-aware floorplanning problem or a parallel version of the genetic algorithm could increase the efficiency of our algorithm. These improvements would allow to study the convergence of the algorithm with an increase in the number of cores. This is a promising study as computer designers tend to place more and more cores in a single die which makes existing thermal hotspots more severe.

Bibliography

[1] Electrothermal monte carlo modelling of submicron hfets. www.nanofolio.org/research/paper03.php, 2004.

[2] Intel tries to keep it cool. www.pcworld.idg.com.au/article/108386/intel tries keep its cool, 2004.

[3] S.N. Adya et al. Fixed-outline floorplanning: enabling hierarchical design. *VLSI Systems, IEEE Transactions on*, 11(6):1120 – 1135, dec. 2003.

[4] ARM. www.arm.com/products/processors/cortex-a/cortex-a9.php.

[5] Johan Berntsson and Maolin Tang. A slicing structure representation for the multi-layer floorplan layout problem. In *EvoWorkshops*, pages 188–197, 2004.

[6] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4), 1999.

[7] G. Chen et al. Partition-driven standard cell thermal placement, 2003.

[8] C. Chu et al. A matrix synthesis approach to thermal placement. *CADICS, IEEE Transactions on*, 17(11):1166 –1174, nov 1998.

[9] C. Coello et al. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer, 2002.

[10] Carlos A. Coello Coello. A short tutorial on evolutionary multiobjective optimization. In *Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization*, EMO '01, pages 21–40, London, UK, 2001. Springer-Verlag.

[11] J. Cong et al. A thermal-driven floorplanning algorithm for 3D ICs. ICCAD, 2004.

[12] Jose L. Ayala David Cuesta, Jose L. Risco and D. Atienza. 3d thermal-aware floorplanner for many-core single-chip systems. *IEEE Latin-American Test Workshop*, 2011.

[13] K. Deb et al. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[14] M. Ekpanyapong et al. Thermal-aware 3d microarchitectural floorplanning. Technical report, Georgia Institute of Technology Center, 2004.

[15] Carlos M. Fonseca and Peter J. Fleming. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In *Genetic Algorithms: Proceedings of the Fifth International Conference*, pages 416–423. Morgan Kaufmann, 1993.

[16] Pei-Ning Guo, T. Takahashi, Chung-Kuan Cheng, and T. Yoshimura. Floorplanning using a tree representation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(2):281 –289, feb 2001.

[17] I. K. Y. Han et al. Temperature aware floorplanning, 2005.

[18] Y. Han et al. Simulated annealing based temperature aware floorplanning, 2007.

[19] M. Healy et al. Multiobjective microarchitectural floorplanning for 2D and 3D ICs. *CADICS, IEEE Transactions on*, 26(1):38–52, 2007.

[20] HPlabs. www.hpl.hp.com/research/cacti/.

[21] W.-L Hung et al. Thermal-aware floorplanning using genetic algorithms. In *ISQED*, pages 634 – 639, march 2005.

[22] China Design Center IBM. Complex soc design, 2009.

[23] S.M.Z. Iqbal et al. ParMiBench - an open-source benchmark for embedded multiprocessor systems. *CAL*, 9(2):45 –48, feb. 2010.

[24] X. Li et al. A novel thermal optimization flow using incremental floorplanning for 3D ICs. In *ASPDAC*, pages 347–352. IEEE Press, 2009.

[25] Y. Li and aothers. Temperature aware floorplanning via geometry programming. In *CSE WORKSHOPS IEEE International Conference on*, pages 295 –298, july 2008.

[26] Y. Liu et al. Accurate temperature-dependent integrated circuit leakage power estimation is easy. In *DATE*, pages 1526–1531, 2007.

[27] OpenSPARC. http://www.opensparc.net/pubs/preszo/07/n2isscc.pdf, 2007.

[28] G. Paci et al. Exploring temperature-aware design in low-power mpsocs. *International journal of embedded systems*, 3(1):43–51, 2007.

[29] K. Sankaranarayanan et al. A case for thermal-aware floorplanning at the microarchitectural level. *JILP*, 7(1):8–16, 2005.

[30] L. Singhal et al. Statistical power profile correlation for realistic thermal estimation. ASP-DAC, pages 67–70. IEEE Computer Society Press, 2008.

[31] M. Tang et al. A memetic algorithm for VLSI floorplanning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 37(1):62–69, 2007.

[32] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, 2001.

ARM

Appendix A

Execution distributions

A.1 30 Cores Architecture

This heterogeneous 30 cores architecture is composed of 20 SPARC, 5 ARM and 5 PPC.

Calculus benchmark: A group of 12 processors (8 SPARC + 2 ARM + 2 PPC) executing bitcounts, a group of 6 processors (4 SPARC + 1 ARM + 1 PPC) solving cubic equations, a group of 6 processors (4 SPARC +1 ARM + 1 PPC) executing deg to rad conversions and a group of 6 processors (4 SPARC + 1 ARM + 1 PPC) executing integer square roots.

Application	SPARC id	ARM id	PPC id
Bitcount	0,1,2,3,4,5,6,7	20,21	25,26
cubic	8,9,10,11	22	27
deg2rad	12,13,14,15	23	28
sqrt	16,17,18,19	24	29

Network benchmark: Three groups of six processors (4 SPARC + 1 ARM + 1 PPC) computing the Dijkstra shortest path algorithm and 12 processors (8 SPARC + 2 ARM + 2 PPC) computing a Patricia algorithm.

Application	SPARC id	ARM id	PPC id
Patricia	0,1,2,3,4,5,6,7	20,21	25,26
dijkstra1	8,9,10,11	22	27
dijkstra2	12,13,14,15	23	28
dijkstra3	16,17,18,19	24	29

Security benchmark: All 30 processors executing the SHA algorithm.

Application	SPARC id	ARM id	PPC id
sha	0,1,...,19	20,21,22,23,24	25,26,27,28,29

Office benchmark: A group of 15 processors (10 SPARC + 3 ARM + 2 PPC) executing string searches using the Boyer-Moore-Horspool method and a group of 15 processors (10 SPARC + 2 ARM + 3 PPC) executing string searches using the Pratt-Boyer-Moore method.

Application	SPARC id	ARM id	PPC id
BMH	0,1,2,3,4,5,6,7,8,9	20,21,22	22,26
PBM	10,11,12,13,14,15,16,17,18,19	23,24	27,28,29

Multimedia benchmark: Three groups of six processors (4 SPARC + 1 ARM + 1 PPC) executing the corner finder algorithm and two groups of six processors (4 SPARC + 1 ARM + 1 PPC) executing the image smoothing algorithm.

Application	SPARC id	ARM id	PPC id
corner1	0,1,2,3	20	25
corner2	4,5,6,7	21	26
corner3	8,9,10,11	22	27
smooth1	12,13,14,15	23	28
smooth2	16,17,18,19	24	29

Mixed benchmark: Three groups of six processors (4 SPARC + 1 ARM + 1 PPC) computing the Dijkstra shortest path algorithm, a group of 6 processors (4 SPARC + 1 ARM + 1 PPC) solving cubic equations and a group of 6 processors (4 SPARC + 1 ARM + 1 PPC) executing deg to rad conversions.

Application	SPARC id	ARM id	PPC id
Dijkstra1	0,1,2,3	20	25
Dijkstra2	4,5,6,7	21	26
Dijkstra3	8,9,10,11	22	27
cubic	12,13,14,15	23	28
deg2rad	16,17,18,19	24	29

A.2 66 Cores Architecture

This heterogeneous 66 cores architecture is composed of 22 SPARC, 22 ARM and 22 PPC.

Calculus benchmark: Two groups of 15 processors (5 SPARC + 5 ARM + 5 PPC) executing bitcounts, 12 processors (4 SPARC +4+4) solving cubic equations, 12 processors (4 SPARC + 4 ARM + 4 PPC) executing deg to rad conversions and 12 processors (4 SPARC + 4 ARM + 4 PPC) executing integer square roots.

Application	SPARC id	ARM id	PPC id
Bitcount	0,1,2,3,4	22,23,24,25,26	44,45,46,47,48
Bitcount2	5,6,7,8,9	27,28,29,30,31	49,50,51,52,53
cubic	10,11,12,13	32,33,34,35	54,55,56,57
deg2rad	14,15,16,17	36,37,38,39	58,59,60,61
sqrt	18,19,20,21	40,41,42,43	62,63,64,65

Network benchmark: Seven groups of six processors (2 SPARC + 2 ARM + 2 PPC) computing the Dijkstra shortest path algorithm and two groups of 12 processors (4 SPARC + 4 ARM + 4 PPC) computing a Patricia algorithm.

Application	SPARC id	ARM id	PPC id
patricia1	0,1,2,3	22,23,24,25	44,45,46,47
patricia2	4,5,6,7	26,27,28,29	48,49,50,51
dijkstra1	8,9	30,31	52,53
dijkstra2	10,11	32,33	54,55
dijkstra3	12,13	34,35	56,57
dijkstra4	14,15	36,37	58,59
dijkstra5	16,17	38,39	60,61
dijkstra6	18,19	40,41	62,63
dijkstra7	20,21	42,43	64,65

Security benchmark: All 66 processors executing the SHA algorithm.

Application	SPARC id	ARM id	PPC id
sha	0,1...21	21,22...43	44,45...65

Office benchmark: A group of 33 processors (11 SPARC + 11 ARM + 11 PPC) executing string searches using the Boyer-Moore-Horspool method and a group of 33 processors (11 SPARC + 11 ARM + 11 PPC) executing string searches using the Pratt-Boyer-Moore method.

Application	SPARC id	ARM id	PPC id
BMH	0,1,2...10	22,23...32	44,45...54
PBM	11,12...21	33,34...43	55,56...65

Multimedia benchmark: Six groups of six processors (2 SPARC + 2 ARM + 2 PPC) executing the corner finder algorithm and five groups of six processors (2 SPARC + 2 ARM + 2 PPC) executing the image smoothing algorithm.

Application	SPARC id	ARM id	PPC id
corner1	0,1	22,23	44,45
corner2	2,3	24,25	46,47
corner3	4,5	26,27	48,49
corner4	6,7	28,29	50,51
corner5	8,9	30,31	52,53
corner6	10,11	32,33	54,55
smooth1	12,13	34,35	56,57
smooth2	14,15	36,37	58,59
smooth3	16,17	38,39	60,61
smooth4	18,19	40,41	62,63
smooth5	20,21	42,43	64,65

Mixed benchmark: Seven groups of six processors (2 SPARC + 2 ARM + 2 PPC) computing the Dijkstra shortest path algorithm, a group of 12 processors (4 SPARC + 4 ARM + 4 PPC) solving cubic equations and a group of 12 processors (4 SPARC + 4 ARM + 4 PPC) executing deg to rad conversions.

Application	SPARC id	ARM id	PPC id
dijkstra1	0,1	22,23	44,45
dijkstra2	2,3	24,25	46,47
dijkstra3	4,5	26,27	48,49
dijkstra4	6,7	28,29	50,51
dijkstra5	8,9	30,31	52,53
dijkstra6	10,11	32,33	54,55
dijkstra7	12,13	34,35	56,57
cubic	14,15,16,17	36,37,38,39	58,59,60,61
deg2rad	18,19,20,21	40,41,42,43	62,63,64,65

A.3 129 Cores Architecture

This heterogeneous 129 cores architecture is composed of 43 SPARC, 43 ARM and 43 PPC

Calculus benchmark: Four groups of 15 processors (5 SPARC + 5 ARM + 5 PPC) executing bitcounts, a group of 23 processors (8 SPARC + 8 ARM + 7 PPC) solving cubic equations, a group of 23 processors (8 SPARC + 7 ARM + 8 PPC) executing deg to rad conversions and a group of 23 processors (7 SPARC + 8 ARM + 8 PPC) executing integer square roots.

Application	SPARC id	ARM id	PPC id
Bitcount1	0,1,2,3,4	43,44,45,46,47	86,87,88,89,90
Bitcount2	5,6,7,8,9	48,49,50,51,52	91,92,93,94,95
Bitcount3	10,11,12,13,14	53,54,55,56,57	96,97,98,99,100
Bitcount4	15,16,17,18,19	58,59,60,61,62	101,102,103,104,105
cubic	20,21..27	63,64..70	106,107..112
deg2rad	28,29..35	71,72..77	113,114..120
sqrt	36,37..42	78,79..85	121,122..128

Network benchmark: Fourteen groups of six processors (2 SPARC + 2 ARM + 2 PPC) computing the Dijkstra shortest path algorithm, three groups of 12 processors (4 SPARC + 4 ARM + 4 PPC) computing a Patricia algorithm and a group of 9 processors (3 SPARC + 3 ARM + 3 PPC) computing a Patricia algorithm.

Application	SPARC id	ARM id	PPC id
patricia1	0,1,2,3	43,44,45,46	86,87,88,89
patricia2	4,5,6,7	47,48,49,50	90,91,92,93
patricia3	8,9,10,11	51,52,53,54	94,95,96,97
patricia4	12,13,14	55,56,57	98,99,100
dijkstra1	15,16	58,59	101,102
dijkstra2	17,18	60,61	103,104
dijkstra3	19,20	62,63	105,106
dijkstra4	21,22	64,65	107,108
dijkstra5	23,24	66,67	109,110
dijkstra6	25,26	68,69	111,112
dijkstra7	27,28	70,71	113,114
dijkstra8	29,30	72,73	115,116
dijkstra9	31,33	74,75	117,118
dijkstra10	33,34	76,77	119,120
dijkstra11	35,36	78,79	121,122
dijkstra12	37,38	80,81	123,124
dijkstra13	39,40	82,83	125,126
dijkstra14	41,43	84,85	127,128

Security benchmark: All 129 processors executing the SHA algorithm.

Application	SPARC id	ARM id	PPC id
sha	0,1..42	43,44..85	86,87..128

Office benchmark: A group of 64 processors (22 SPARC + 21 ARM + 21 PPC) executing string searches using the Boyer-Moore-Horspool method and 65 processors (21 SPARC + 22 ARM + 22 PPC) executing string searches using the Pratt-Boyer-Moore method.

Application	SPARC id	ARM id	PPC id
BMH	0,1,2,3	43,44..63	86,87..106
PBM	4,5,6,7	64,65..85	107,108..128

Multimedia benchmark: Eleven groups of six processors (2 SPARC + 2 ARM + 2 PPC) executing the corner finder algorithm, a group of 3 processors (1 SPARC + 1 ARM + 1 PPC) executing the corner finder algorithm and ten groups of six processors (2 SPARC + 2 ARM + 2 PPC) executing the image smoothing algorithm.

Application	SPARC id	ARM id	PPC id
corner1	0,1,	43,44	86,87
corner2	2,3	45,46,	88,89
corner3	4,5	47,48,	90,91
corner4	6,7	49,50	92,93
corner5	8,9	51,52	94,95
corner6	10,11	53,54	96,97
corner7	12,13	55,56	98,99
corner8	14,15	57,58	100,101
corner9	16,17	59,60	102,103
corner10	18,19	61,62	104,105
corner11	20,21	63,64	106,107
corner12	22	65	108
smooth1	23,24	66,67	109,110
smooth2	25,26	68,69	111,112
smooth3	27,28	70,71	113,114
smooth4	29,30	72,73	115,116
smooth5	31,33	74,75	117,118
smooth6	33,34	76,77	119,120
smooth7	35,36	78,79	121,122
smooth8	37,38	80,81	123,124
smooth9	39,40	82,83	125,126
smooth10	41,42	84,85	127,128

Mixed benchmark: Fourteen groups of six processors (2 SPARC + 2 ARM + 2 PPC) computing the Dijkstra shortest path algorithm, a group of 15 processors (5 SPARC + 5 ARM + 5 PPC) solving cubic equations, a group of 15 processors (5 SPARC + 5 ARM + 5 PPC) executing deg to rad conversions and a group of 15 processors (5 SPARC + 5 ARM + 5 PPC) executing executing integer square roots.

Application	SPARC id	ARM id	PPC id
dijkstra1	0,1,	43,44	86,87
dijkstra2	2,3	45,46	88,89
dijkstra3	4,5	47,48	90,91
dijkstra4	6,7	49,50	92,93
dijkstra5	8,9	51,52	94,95
dijkstra6	10,11	53,54	96,97
dijkstra7	12,13	55,56	98,99
dijkstra8	14,15	57,58	100,101
dijkstra9	16,17	59,60	102,103
dijkstra10	18,19	61,62	104,105
dijkstra11	20,21	63,64	106,107
dijkstra12	22,23	65,66	108,109
dijkstra13	24,25	67,68	110,111
dijkstra14	26,27	69,70	112,113
cubic	28,29,30,31,32	71,72,73,74,75	114,115,116,117,118
deg2rad	33,34,35,36,37	76,77,78,79,80	119,120,121,122,123
sqrt	38,39,40,41,42	81,82,83,84,85	124,125,126,127,128

Appendix B

Scheduling algorithm

For example, Figure B.1 shows our multiprocessor scheduling algorithm used to simulate an heterogeneous platform composed of 20 SPARC, 5 ARM and 5 PPC processors.

$simTimeSlice = 0.016$
$instPerTimeSliceSPARC = 120000000 * simTimeSlice$; $instPerTimeSliceARM = 80000000 * simTimeSlice$;
$instPerTimeSlicePPC = 60000000 * simTimeSlice$;
$nextWindow = 0.128$; $maxSim = 15.0$; $windowId = 0$;
for $myTime = 0 \rightarrow maxSim$ **do**
  for $i = 0 \rightarrow 20$ **do**
    $instBefore = icmGetProcessorICount(processor[i])$
    $rtnVal[i] = icmSimulate(processor[i], instPerTimeSliceSPARC)$
    $instExe = icmGetProcessorICount(processor[i]) - instBefore$; $instrWindow[i] += instExe$
    $i \leftarrow i + 1$
  end for
  for $i = 20 \rightarrow 25$ **do**
    $instBefore = icmGetProcessorICount(processor[i])$
    $rtnVal[i] = icmSimulate(processor[i], instPerTimeSliceARM)$
    $instExe = icmGetProcessorICount(processor[i]) - instBefore$; $instrWindow[i] += instExe$
    $i \leftarrow i + 1$
  end for
  for $i = 25 \rightarrow 30$ **do**
    $instBefore = icmGetProcessorICount(processor[i])$
    $rtnVal[i] = icmSimulate(processor[i], instPerTimeSlicePPC)$
    $instExe = icmGetProcessorICount(processor[i]) - instBefore$; $instrWindow[i] += instExe$
    $i \leftarrow i + 1$
  end for
  if $myTime == next_window$ **then**
    $writeStats()$
    $windowId \leftarrow windowId + 1$; $nextWindow \leftarrow myTime + 0.128$
  end if
  $myTime \leftarrow myTime + simTimeSlice$; $icmAdvanceTime(myTime)$
end for

Figure B.1: *Multiprocessor Scheduling algorithm*